VISIT…

Beginning Java™ ME Platform

Ray Rischpater

Apress®

Beginning Java™ ME Platform

Copyright © 2008 by Ray Rischpater

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1061-0

ISBN-10 (pbk): 1-4302-1061-3

ISBN-13 (electronic): 978-1-4302-1062-7

ISBN-10 (electronic): 1-4302-1062-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editor: Steve Anglin
Technical Reviewer: Christopher King
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Nicole Abramowitz
Associate Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Patrick Cunningham
Proofreader: Liz Welch
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

About the Author xvii
About the Technical Reviewer xix
Acknowledgments xxi
Introduction xxiii

PART 1 Getting Started

CHAPTER 1 Mapping the Java Jungle 3
CHAPTER 2 Shrinking Java to Fit 19
CHAPTER 3 Getting Started with the NetBeans IDE 33

Intermezzo

PART 2 CLDC Development with MIDP

CHAPTER 4 Introducing MIDlets 83
CHAPTER 5 Building User Interfaces 97
CHAPTER 6 Storing Data Using the Record Store 133
CHAPTER 7 Accessing Files and Other Data 161
CHAPTER 8 Using the Java Mobile Game API 193

Intermezzo

PART 3 CDC Development

CHAPTER 9 Introducing Xlets and the Personal Basis Profile 223
CHAPTER 10 Introducing Applets and the Advanced Graphics and User Interface 253
CHAPTER 11 Using Remote Method Invocation 273

Intermezzo

PART 4 Communicating with the Rest of the World

CHAPTER 12 Accessing Remote Data on the Network 293
CHAPTER 13 Accessing Web Services 331
CHAPTER 14 Messaging with the Wireless Messaging API 373

Intermezzo

PART 5 Other Java ME Interfaces

CHAPTER 15 Securing Java ME Applications 413
CHAPTER 16 Rendering Multimedia Content 447
CHAPTER 17 Finding Your Way 499
CHAPTER 18 Seeking a Common Platform 523
APPENDIX Finding Java APIs 539

INDEX 543

Contents

About the Author xvii
About the Technical Reviewer xix
Acknowledgments xxi
Introduction xxiii

PART 1 Getting Started

CHAPTER 1 Mapping the Java Jungle 3

Introducing the Market for Java ME 3
Looking from the Device Manufacturers' Perspective 3
Looking from the Operators' Perspective 4
Looking from the Consumers' Perspective 5
Looking Inside the Java ME Platform 6
Justifying the Need for a Mobile Edition of Java 6
Making Java Work on Mobile Devices 7
Understanding Configurations 10
Introducing the Connected Limited Device Configuration 10
Introducing the Connected Device Configuration 12
Understanding Profiles 12
Introducing the Mobile Information Device Profile 13
Introducing the Foundation Profile 14
Introducing the Personal Basis Profile 14
Introducing the Personal Profile 15
Understanding Packages 15
Planning Your Approach to Java ME Development 16
Selecting Appropriate Device Targets 16
Marketing and Selling Your Application 17
Wrapping Up 18

CHAPTER 2 Shrinking Java to Fit 19

Making It Fit: The CLDC 19
Understanding the Present: CLDC 1.1 20
Looking Back at CLDC 1.0 22
Looking Toward the Future of the CLDC 22
Making It Fit: The CDC 23
Changing the Java Class Library to Fit the CLDC 24
Changes to the java.lang Package 24
Changes to the java.util Package 28
Changes to the java.io Package 29
Introducing Classes in the CLDC 30
Changing the Java Class Library to Fit the CDC 31
Wrapping Up 31

CHAPTER 3 Getting Started with the NetBeans IDE 33

Selecting the NetBeans IDE 33
Finding Your Way Around the NetBeans IDE 35
Creating Your First CLDC/MIDP Application 37
Walking Through the Creation of WeatherWidget 38
Building CLDC/MIDP Applications 52
Packaging and Executing CLDC/MIDP Applications 53
Creating Your First CDC Application 57
Walking Through the Creation of WeatherApplet 57
Packaging and Executing CDC Applications 75
Wrapping Up 77

Intermezzo

PART 2 CLDC Development with MIDP

CHAPTER 4 Introducing MIDlets 83

Looking at the Simplest MIDlet 83
Understanding the MIDlet Life Cycle 85
Packaging MIDlets 87
Obtaining Properties and Resources 89
Managing Startup Events and Alarms 90
Wrapping Up 96

CHAPTER 5 Building User Interfaces 97

Understanding the Relationship Between the Display and Visible Item Objects 97
Using Commands to Control Application Flow 101
Introducing Basic Visible Items 104
Introducing Items 106
Managing Choices 112
Introducing the Screen and Its Subclasses 114
Collecting Visible Items Using the Form Class 114
Alerting the User 116
Accepting Copious Amounts of Text 119
Showing Lists of Choices 120
Working with the Canvas and Custom Items 122
Controlling Drawing Behavior with a Custom Canvas 122
Creating a Custom Item for a Screen 125
Implementing a Custom Item 127
Wrapping Up 131

CHAPTER 6 Storing Data Using the Record Store 133

Peeking Inside the Record Store 133
Using the Record Store 135
Opening and Closing a Record Store 136
Removing a Record Store 137
Obtaining Information About a Record Store 137
Accessing Records in the Record Store 138
Adding a Record 141
Retrieving a Record 142
Enumerating a Record 142
Updating a Record 144
Removing a Record 144
Counting Records 145
Listening for Record Store Changes 145
Understanding Platform Limitations of Record Stores 145
Putting the Record Store to Work 146
Wrapping Up 160

CHAPTER 7 Accessing Files and Other Data 161

Introducing the FCOP 161
Using the FCOP 163
Determining If the FCOP Is Present 164
Obtaining a FileConnection Instance 164
Creating a New File or Directory 165
Opening a File 166
Tweaking File Attributes 166
Deleting a File or Directory 167
Enumerating a Directory's Contents 167
Listening for File System Changes 168
Putting the FCOP to Work 169
Introducing the PIM Package 174
Using the PIM Package 175
Ensuring the PIM Package Is Available 176
Opening a PIM Database 176
Reading Records from a PIM Database 177
Reading Fields from a PIM Record 177
Modifying a PIM Record 182
Adding a PIM Record 183
Removing a PIM Entry 184
Managing PIM Database Categories 184
Putting the PIM Package to Work 185
Understanding the Role Code Signing and Verification Can Play 190
Wrapping Up 191

CHAPTER 8 Using the Java Mobile Game API 193

Looking Inside the Mobile Game API 193
Managing Events and Drawing 195
Polling for Keystrokes 196
Managing Game Execution 197
Tying Your GameCanvas to Your MIDlet 199
Layering Visual Elements 200
Managing Layers 201
Optimizing Visual Layers Using Tiling 202
Producing Animations 205

Putting the Mobile Game API to Work 207
Implementing the Game MIDlet 209
Implementing the Game Canvas 210
Wrapping Up 218

Intermezzo

PART 3 CDC Development

CHAPTER 9 Introducing Xlets and the Personal Basis Profile 223

Understanding the Xlet 223
Looking at the Xlet Life Cycle 224
Extending the Xlet Interface 225
Using the Xlet Context 226
Writing a Simple Xlet 227
Looking at a Simple Xlet 227
Understanding Xlet Dependencies 230
Developing Lightweight User Interfaces Using the PBP 233
Implementing Your Own Components for a Window Toolkit 234
Writing a Simple, Lightweight Component 236
Understanding Window Toolkit Limitations of the PBP 240
Obtaining Xlet Properties and Resources 242
Communicating with Other Xlets 243
Implementing a Shared Object 244
Sharing an Object for Other Xlets to Find 246
Using a Shared Object 249
Wrapping Up 251

CHAPTER 10 Introducing Applets and the Advanced Graphics and User Interface 253

Writing Applets for Java ME 253
Looking at the Applet Life Cycle 254
Presenting the Applet's User Interface 256
Accessing an Applet's Context 257
Communicating Between Applets 258

Developing User Interfaces with the AWT 260
Using AWT Containers 262
Using AWT Components 263
Handling AWT Events 264
Developing User Interfaces with the AGUI 266
Understanding Restrictions on Top-Level Windows 269
Using the AGUI's Added Input Support 269
Understanding Changes to the Drawing Algorithm 270
Wrapping Up 271

CHAPTER 11 Using Remote Method Invocation 273

Understanding Java RMI 273
Understanding the Architecture of Java RMI 274
Introducing the Java RMI Interfaces 277
Understanding the Java RMI Optional Package 278
Looking at the Requirements for the Java RMI Optional Package 278
Seeing What's Provided by the Java RMI Optional Package 279
Applying Java RMI 280
Writing the Java Interfaces for the Service 282
Implementing the Service Using Java SE 283
Generating the Stub Classes for Java SE 284
Writing the Remote Service Host Application 285
Invoking the Remote Object from the Client 286
Wrapping Up 286

Intermezzo

PART 4 Communicating with the Rest of the World

CHAPTER 12 Accessing Remote Data on the Network 293

Introducing the Generic Connection Framework 293
Communicating with Sockets and Datagrams 300
Using Sockets with the GCF 300
Using Datagrams with the GCF 304
Communicating with HTTP 306
Reviewing HTTP 306
Using HTTP with the GCF 309
Putting HTTP to Work 315
Securing Your HTTP Transaction with HTTPS 325
Granting Permissions for Network Connections 327
Wrapping Up 328

CHAPTER 13 Accessing Web Services 331

Looking at a Web Service from the Client Perspective 331
Considering the Architecture 333
Exchanging Data over the Network 334
Using XML for Data Representation 336
Exploring XML Support for Web Services in Java ME 341
Generating XML in Java ME Applications 343
Introducing the J2ME Web Services Specification 355
Introducing the kXML Parser 365
Wrapping Up 372

CHAPTER 14 Messaging with the Wireless Messaging API 373

Introducing Wireless Messaging Services 373
Introducing Short Message Service 374
Introducing Multimedia Messaging Service 374
Introducing the Cell Broadcast Service 375

Introducing Wireless Messaging API 375
Creating Messages 379
Sending Messages 380
Receiving Messages 385
Managing Message Headers 385
Understanding Required Privileges When Using the WMA 386
Using the Push Registry 387
Registering Dynamically for Incoming Messages. 390
Using PushRegistry APIs. 390
Applying the Wireless Messaging API 391
Sending and Receiving SMS Messages. 391
Sending and Receiving MMS Messages 398
Wrapping Up. 407

Intermezzo

PART 5 Other Java ME Interfaces

CHAPTER 15 Securing Java ME Applications 413

Understanding the Need for Security 413
Looking at Java ME's Security and Trust Services 416
Communicating with Cryptographic Hardware Using the APDU API 417
Communicating with Java Smart Cards Using JCRMI. 420
Leveraging the SATSA High-Level APIs for Cryptography. 422
Exploring the Bouncy Castle Solution to Security Challenges 425
Creating Message Digests Using the Bouncy Castle API 428
Encrypting and Decrypting Using the Bouncy Castle API 429
Creating Secure Commerce with Contactless Communications. 431
Discovering Contactless Targets. 432
Communicating with Contactless Targets. 435
Recognizing and Generating Visual Tags. 440
Wrapping Up. 444

CHAPTER 16 Rendering Multimedia Content 447

Introducing the MMAPI 448
Understanding Basic Multimedia Concepts 448
Understanding the Organization of the MMAPI 450
Starting the Rendering Process 454
Controlling the Rendering Process 458
Capturing Media 461
Playing Individual Tones 466
Introducing the Java Scalable 2D Vector Graphics API 470
Understanding Basic SVG Concepts 470
Understanding the Organization of the SVGAPI 472
Rendering SVG Images 474
Modifying SVG Images 480
Using NetBeans with SVG Images 483
Putting the MMAPI and the SVGAPI to Work 484
Playing Audio and Video 493
Capturing Images 494
Playing SVG Content 496
Wrapping Up 497

CHAPTER 17 Finding Your Way 499

Understanding Location-Based Services 499
Introducing the Location API 501
Understanding the Location API 502
Using the Location API to Determine Device Location 503
Using the Location API to Manage Landmarks 507
Understanding the Role That Security Plays in LBS 508
Using the Location API 509
Locating the User 518
Simulating Location API Data in the Sun Java Wireless Toolkit 518
Wrapping Up 520

CHAPTER 18 Seeking a Common Platform 523

Understanding the Role JSRs Play in Fragmentation 523
Contributing to Fragmentation and Unification 524
Reading a JSR 525
Dealing with Fragmentation on Your Own 527
Understanding the JTWI 528
Examining the JTWI Required Elements 529
Examining the JTWI Optional Elements 529
Understanding the MSA 530
Understanding MSA 1.0 531
Evolving for the Future: MSA2 534
Wrapping Up 537

APPENDIX Finding Java APIs 539

INDEX 543

About the Author

RAY RISCHPATER is an engineer and author with more than 15 years of experience writing about and developing for mobile-computing platforms. During this time, Ray has participated in the development of Internet technologies for Java ME, Qualcomm BREW, Palm OS, Apple Newton, and General Magic's Magic Cap, as well as several proprietary platforms. Presently, Ray is employed as the chief architect at Rocket Mobile, a wholly owned subsidiary of Buongiorno Group. When not writing for or about mobile platforms, Ray enjoys hiking with his family and participating in public service through amateur radio in and around the San Lorenzo Valley in northern California. Ray holds a bachelor's degree in pure mathematics from the University of California, Santa Cruz and is a member of the Institute of Electrical and Electronics Engineers (IEEE), the Association for Computing Machinery (ACM), and the American Radio Relay League (ARRL). Ray's previous books include *Software Development for the QUALCOMM BREW Platform* (Apress, 2003), *Wireless Web Development, Second Edition* (Apress, 2002), and *eBay Application Development* (Apress, 2004).

About the Technical Reviewer

CHRIS KING has been writing software since childhood; today he focuses on the challenges and joys of mobile development. In recent years, he has specialized in technologies such as Java ME, Qualcomm BREW, and Android. His recent projects include messaging software that has been preloaded on millions of phones, consumer entertainment devices, middleware libraries, community organizing tools, and lifestyle applications. Chris currently serves as a lead engineer for Gravity Mobile in San Francisco.

Since moving to California, Chris has become an avid hiker, cyclist, and home cook. With any free time that remains, Chris programs for fun, writes, and devours books.

Acknowledgments

Any book today is the collaborative effort of numerous people; technical books such as this one even more so. In helping me produce this book, I owe thanks to numerous people, including some who don't realize how much they helped, and others whose names I may never know.

My son Jarod has been part of my writing career since it started; my first book and his birth nearly coincided. He is now old enough that he is writing both prose and programs on his own, giving us valuable opportunities to share in learning together. His respect for the craft of writing—shown through his asking me questions about what I am doing and how I do it—is precious to me. His ability to help me wholly forget the frustrations inherent in any large project when he and I are together is just one of the many priceless gifts he gives me.

My wife Meg embraced and encouraged this project from the beginning, despite knowing it would mean that I would spend countless hours apart from her as I researched and wrote the examples and text for this book. Her patience with my absence—extending to when I was physically present yet mumbling about some minutia of mobile-application development—bordered on the heroic at times. I cherish our relationship, and it moves me to reflect how each of us supports the other to grow and succeed.

The entire Apress staff was indispensible in bringing this book to you. Steve Anglin and Richard Dal Porto were crucial in helping start the project and shepherd it to completion. Richard was especially helpful in keeping all of the different parts of production running smoothly, even when I found myself missing the occasional deadline. Nicole Abramowitz, my copy editor, was both thorough and patient, and made innumerable improvements to this book. Katie Stence, my production editor, made the production review process painless as I saw how the book would appear in print for the first time. I also must thank those at Apress whom I have not met personally, because without their contributions, Apress would not be the successful company with which I find it so easy to work.

Chris King, this book's technical editor, is also my colleague and friend. His attention to detail frequently transcended errors of program syntax and improved my exposition of many of the concepts you encounter in this book. He fearlessly ran—and *read*—every example in this book and helped improve even the pseudocode that I use in many places as examples. I have always enjoyed working with Chris professionally, and this project cements my professional respect for him.

My colleagues at Rocket Mobile (now part of Buongiorno Group) deserve recognition not only for providing additional Java ME experience on which to draw for several examples in the book, but also for their patience and support. I must apologize to Erik Browne, Levon Dolbakian, Graham Darcy, Jonathan Jackson, and Rajiv Ramanasankaran for enduring my frequent C and Java transpositions as I wrote C code for the office during the day and Java code for the book at night. The management staff—including Young Yoon, Scott Sumner, Jim Alisago, and Wayne Yurtin—has given me the privilege of combining software engineering and writing, and has provided a climate in which both can succeed. Thank you, each and every one of you.

Many people close to me contributed additional support, whether or not they knew they were doing so. Brad Holden, Connie Rockosi, Chris Haseman, and Shane Conder are at the top of this list for giving me much-needed space, time to work, and positive encouragement during the many times when I wondered if it were possible to write a book while working full-time and having an active life outside my technical career. I am indebted to these and others for their contributions as well.

Introduction

When I set out to write this book, I was often surprised by the comments I received from friends and colleagues. Many asked me if some other platform, such as Android or the iPhone, would render Java Platform, Micro Edition (Java ME) obsolete (and non-existent, some posited) by the time the book is published. Still others pointed to the growing convergence between different lines of Java as rendering the need for separate information about Java ME obsolete. And a few remarked scathingly that the market for Java books was saturated, so investing the time to write another was an exercise in futility. You, too, may ask these questions as you decide whether or not to read this book. Perhaps you're interested in Java ME as a specific platform on which to deploy an existing product, or perhaps you're just curious as to whether you should include Java ME skills in your professional portfolio.

The Java ME platform is a highly successful one. *Billions*—yes, that's with a *b*—of devices that run Java ME are in the hands of consumers right now. Still more are on the way, including mobile phones, set-top boxes, and other devices you can't even imagine that are now in development. Java ME is deeply entrenched in the market, and yet through the Java Community Process (JCP), it evolves rapidly to address challenges raised by existing and new competing platforms, including Qualcomm BREW, Android, and the Apple iPhone.

The cross-pollination between Java ME, Java Platform, Standard Edition (Java SE), and Java Platform, Enterprise Edition (Java EE) is well recognized and will continue. Members of the JCP work carefully to introduce APIs that can be shared across these Java platforms, and many Java ME APIs are subsets of APIs proposed or developed for Java SE. In some cases, the opposite is true: Java ME APIs are being introduced into Java SE, such as the Java ME framework for communications and networking. As devices become more capable, you will see more convergence between the various Java lines, but the specific constraints on mobile devices—including ubiquitous network access, a small form factor, and scarce power, memory, and processor resources—will drive the need for specific accommodations within the Java platform. Java ME and the JCP provide a framework for vendors to make those accommodations.

There are many excellent books about Java 2 Platform, Micro Edition (J2ME)—the predecessor to Java ME—and several good books about facets of Java ME as well. However, the Java ME platform evolves and advances at a truly awe-inspiring rate, and this fact and the sheer size of Java ME make it difficult to find a good book for beginners that provides a broad foundation on which to build Java ME competency. In this book, I've worked to balance the presentation of the two profiles that comprise Java ME, because I

believe that for you to be successful, you need to understand both. At the same time, I've made explicit choices about the required and optional Java ME APIs I present, because I believe that in building this foundation, you need to understand some basic principles that arise again and again in the Java ME world, but you don't necessarily need to be able to recall from memory every method from every optional Java ME class. Given the time you have, I believe it is important for you to master the platform fundamentals, so that you're better equipped to specialize in the areas that interest you later. In short, what I *don't* present here may be as important to you as what I *do* present.

Why Should You Read This Book?

I've already partially answered this question, but it's worth recapping: Java ME is an integral part of the mobile-computing marketplace, and it's a platform that every software developer who works with mobile devices should be familiar with. Whether you need to use it daily in your job, see it as competition, or are simply curious about how it's different from the platforms for which you presently develop applications, understanding Java ME fundamentals will make you a better mobile software developer.

Whether you're new to mobile-application development or have written mobile applications for other platforms and are interested in learning what you need to know to be a Java ME developer, you should read this book. By turning equal attention to the two Java ME configurations—the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC)—I prepare you to write software for either the booming mobile-phone market or the nascent market for set-top boxes and high-end mobile phones with advanced user interfaces and other capabilities. Because Java ME devices at their core are network-enabled devices, I spend a great deal of time explaining to you the APIs that Java ME uses to enable applications to communicate, and I prepare you to understand new communication schemes that Java ME may use in the coming years. Once you finish this book, you can expect to have a grasp of the most important APIs that Java ME developers use, as well as an understanding of the fundamental thinking behind the design and approach of the Java ME platform and the dynamics of the mobile-software marketplace as a whole.

However, I have some expectations of you as well. I assume you have at least some previous exposure to Java SE—both the language and some of the major classes that it supports. You may not know the difference between a `HashMap` and a `TreeMap`, but you should at least have a nodding acquaintance with Java syntax, the Java package system, and some of the basic foundation classes that you can find in the `java.lang` and `java.util` packages. Because it's an important communication tool, you should also have at least a nodding acquaintance with Unified Modeling Language (UML), as I frequently use UML class, state machine, and sequence diagrams to help illustrate the relationship between various Java ME components.

Don't worry, though, if you're new to mobile-software development. One primary aim of this book is to help you understand the dynamics of the mobile software–development marketplace, because those dynamics have and continue to influence Java ME. I firmly believe that a good software developer understands not just the platform, but the business behind the market as well. I also don't expect you to be a Java expert: you can write solid code clearly using a minimum of Java-specific language features. If I throw a closure or anonymous inner class your way, I'll let you know; my goal here is for you to learn to write mobile applications, not become the office Java guru.

In the interest of full disclosure, there may be reasons why this book isn't for you. I don't discuss every optional Java ME API in detail—for example, I omit discussions of both the Java Mobile 3D Graphics API and Java ME support for Bluetooth—because they're well covered by other texts and because they're not necessary material that every Java ME developer must know. In a similar vein, if you already have a great deal of Java ME experience under your belt, you may still learn something from this book, but your time may be better spent with a more in-depth exploration of a specific set of optional APIs that interest you. For example, another source, such as a Java Specification Request (JSR) that describes a particular API or a book on a specific topic, may be better for you. I intend this book to be a survey for beginners new to the platform that calls out the rules of the road and relevant landmarks, not an atlas of every intersection, hilltop, creek, island, and bay.

How Should You Read This Book?

Presenting Java ME to newcomers poses particular challenges, because in many ways, Java ME is really two platforms: one that's wildly successful for mobile phones, and a second that's deployed in other consumer-electronics markets. As an engineer myself, I recognize how busy you are and how you may be looking to me to give you only the information you need to solve a set of problems on a specific platform, such as a set-top box running the Java ME CDC. Consequently, I've split this book into five parts, so that you can pick and choose the information that's relevant to you.

- *Part 1, "Getting Started"*: Exposes you to the information that every Java ME developer should know: how Java ME is organized, which APIs are common across all Java ME platforms, and which tools are available. I strongly recommend you read the three chapters in this part to orient yourself to the Java ME market and mindset.

- *Part 2, "CLDC Development with MIDP"*: Explores the Java ME Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP) in detail. This configuration and associated profile comprise the most widely deployed mobile-application platform in the world, and if you're interested in writing software for mobile phones or other wireless terminals, you'll need to have a good grasp of what it offers.

- *Part 3, "CDC Development"*: Explores the Java ME Connected Device Configuration (CDC), which underpins many consumer devices today, including television set-top boxes and some advanced mobile phones. The CDC even plays a part in the Blu-ray Disc standard. The information you'll find here is often overlooked in other introductory Java ME materials, but it plays an increasing role in Java ME development.

- *Part 4, "Communicating with the Rest of the World"*: Explains how Java ME enables the applications that you write to communicate with the rest of the Web. You'll learn about the Generic Connection Framework (GCF)—a key addition to the Java world—as well as how Java ME enables you to work with both Internet protocols and wireless-messaging protocols.

- *Part 5, "Other Java ME Interfaces"*: Shows you a few optional APIs that every Java ME developer should know about. These interfaces are important for you to understand both because they provide capabilities nearly every application will tap (such as security and trust interfaces), and because the interfaces provide a fundamental framework that other optional Java APIs extend (such as the Mobile Media API). This part closes with a chapter examining how optional APIs fragment the Java ME platform and how the Java community works together to address this fragmentation.

A short "Intermezzo" precedes each part, helping orient you in the book.
Eighteen chapters await you in the five parts:

- *Chapter 1, "Mapping the Java Jungle"*: Introduces some key vocabulary and business concepts you must understand before becoming a Java ME developer.

- *Chapter 2, "Shrinking Java to Fit"*: Describes the key transformation Java undergoes between Java SE and Java ME. If you're a seasoned Java SE developer, you should read this chapter closely, as it tells you which language features and classes you already know that are available to you in Java ME. If you're fairly new to Java, you should skim this chapter, but don't be worried if you have to flip back to it occasionally.

- *Chapter 3, "Getting Started with the NetBeans IDE"*: Enables you to build your first Java ME applications using the leading software development kit (SDK) for Java ME development. You'll learn why NetBeans is the environment of choice for developing Java ME applications, and you'll learn how to build two simple applications from scratch using NetBeans. These sample applications are the starting points for many of the examples in subsequent chapters. Even if you decide later to switch to another SDK, this chapter will help you understand how the development tools for Java ME fit together. In the process, you'll also get a quick overview of the major features of Java ME as you build these simple applications.

- *Chapter 4, "Introducing MIDlets"*: Begins your exploration of one of the software world's most successful application platforms. You'll learn about the MIDlet, which is the unit of application execution on most Java ME devices.

- *Chapter 5, "Building User Interfaces"*: Describes the hierarchy of user-interface components that are available only to Java ME developers. You'll learn how the Java ME–provided components work and interact, as well as how to extend the Java ME component hierarchy.

- *Chapter 6, "Storing Data Using the Record Store"*: Describes the Java ME record-store model that your applications can use for persistent storage. The record store is available even on devices without a traditional file system, and it gives you the ability to store records of similar data in a searchable, persistent manner.

- *Chapter 7, "Accessing Files and Other Data"*: Provides your first exposure to an optional Java ME API—that is, an API that may not be available on all platforms. It is such an important API, however, that it's one you should master early. You'll need to understand how it and the record-store model presented in the previous chapter work.

- *Chapter 8, "Using the Java Mobile Game API"*: Describes the Java Mobile Game API and shows you how to write simple platform-independent games using Java ME. Game development is a complex subject; rather than get bogged down in details about game development that may not interest some readers, I emphasize the fundamentals of Java ME as they interrelate with game-development concerns.

- *Chapter 9, "Introducing Xlets and the Personal Basis Profile"*: Describes the parts of Java ME that to date have largely applied to fixed consumer electronics, such as set-top boxes. You'll learn about the application model these devices support, as well as the interfaces they offer.

- *Chapter 10, "Introducing Applets and the Advanced Graphics and User Interface"*: Describes additional execution models available on Java ME platforms, plus support for legacy Java applets and an adaptation of Swing available on some Java ME devices.

- *Chapter 11, "Using Remote Method Invocation"*: Shows you how some Java ME devices can use Remote Method Invocation (RMI) to interact with other Java-provided services on the network.

- *Chapter 12, "Accessing Remote Data on the Network"*: Begins your foray into the communication framework supported by all Java ME devices, and shows you how to use it with Internet protocols to access data and services over the network.

- *Chapter 13, "Accessing Web Services"*: Builds on what you learn in Chapter 12 to show you how Java ME's optional APIs and open source packages enable your applications to access web services using Extensible Markup Language (XML) and HTTP.

- *Chapter 14, "Messaging with the Wireless Messaging API"*: Shows you how to use the wireless messaging interfaces available on many Java ME devices. These interfaces enable you to send and receive messages with protocols such as Short Message Service (SMS).

- *Chapter 15, "Securing Java ME Applications"*: Looks at optional Java ME interfaces that provide extensions such as cryptography and access to smart cards, as well as interfaces that enable mobile commerce, such as the optional API for reading radio-frequency identification (RFID) cards and bar codes.

- *Chapter 16, "Rendering Multimedia Content"*: Describes Java ME's approach to providing support for multimedia content rendering. I show you both the Mobile Media API that Java ME devices may provide, as well as an optional API for displaying and animating Scalable Vector Graphics (SVG) images.

- *Chapter 17, "Finding Your Way"*: Describes the optional Java ME interfaces that let your application determine the device location.

- *Chapter 18, "Seeking a Common Platform"*: Closes the book with a discussion of how the optional APIs that Java ME devices may provide challenge application developers like you to find sufficient devices that provide the features your applications require. I also explain how the Java community is addressing that challenge through additional device profiles such as the Java Technology for the Wireless Industry and Mobile Service Architecture (MSA).

- *Appendix, "Finding Java APIs"*: Provides you with a table of interesting mobile technologies and the JSRs that define support for those technologies. When you're finished reading this book and want to learn more about a specific technology and how it interacts with Java ME, you can use this table to determine where to start your research.

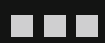
Ideally, I'd encourage you to read all of Parts 1–4 and then whatever parts of Part 5 interest you, especially if this is your first exposure to Java ME. However, you can tackle this material in other ways as well. If you're interested in a specific Java ME configuration, you can first read Part 1, then either Part 2 or Part 3, and then Part 4 and parts of Part 5, for example. Regardless, because some material requires you to master the material that precedes it, you should read material earlier in the book even if you skip around before you dive in to material that comes later in the book.

How Do You Get Started?

Of course, sample applications in this book are all available electronically at the Apress web site, `http://www.apress.com`. Begin by reading Chapters 1 and 2, and then download the NetBeans SDK at `http://www.netbeans.org`; if you're really in a hurry, download the SDK *now* and work through Chapter 3, so you can get a feel for what Java ME application development is all about.

I encourage you to build on what you learn here by consulting other sources; one excellent source is the Java Community Process web site at `http://www.jcp.org`, where you can find the JSRs that describe the Java ME platform (and other Java platforms and extensions to Java platforms as well). If you prefer working on the bleeding edge, the wiki for NetBeans at `http://wiki.netbeans.org` is another excellent resource, especially if you find yourself enamored with the NetBeans environment. Finally, I'll make more resources available as necessary on my web site at `http://www.lothlorien.com`.

PART 1



Getting Started

Before you begin writing code for Java Platform, Micro Edition (Java ME), you should have a good grip on the fundamentals. That's what this part is all about: helping you get a handle on why Java ME is relevant, how to start out writing code for Java ME using NetBeans, and how Java ME differs from traditional Java programming.

CHAPTER 1

Mapping the Java Jungle

Although at its heart Java ME is really just an adaptation of the Java language, class libraries, and concepts to fit constrained devices, the business behind Java ME is in fact quite different. A firm grasp of the Java ME market, platform, and terminology will put you in good stead to developing successful products using Java ME.

In this chapter, I begin by introducing the market for Java ME. Next, I take you on a tour of the Java ME platform, showing you how Sun identified and defined the basic requirements for mobile platforms, and how manufacturers, carriers, and others have extended this basic platform. Finally, I discuss how the process of application development for Java ME is different, and I show you how important it is to know your audience, target devices, and distribution channel.

After reading this chapter, you will understand why Java ME differs from Java. You will see how device manufacturers, wireless operators, and consumers view Java ME, and how Java ME meets the needs of all of these parties. Armed with this knowledge, you'll be able to better manage a Java ME development project.

Introducing the Market for Java ME

A trio of forces dominates the Java ME market: *device manufacturers* looking to differentiate their products in the marketplace, *wireless operators* seeking to differentiate services and raise the average revenue per user (ARPU), and *consumers* personalizing their devices in new and novel ways.

Looking from the Device Manufacturers' Perspective

The interplay between device manufacturers and wireless operators is complex. Manufacturers are in constant competition with each other to differentiate their products, while at the same time, in many markets they are beholden to wireless operators to meet stringent requirements for features and functionality.

Device manufacturers can be broadly separated into two categories: *original equipment manufacturers* (OEMs) and *original design manufacturers* (ODMs). OEMs build

devices under their own label and sell devices to consumers (either directly or via the operator, or most often both), while ODMs design and build hardware on behalf of others. While both ODMs and OEMs must differentiate their product on the basis of price, quality, and features, for OEMs brand and marketing also become key concerns.

Many of today's wireless operators simply require a Java ME runtime on most of the phones that they provide to subscribers. As I discuss in the next section, "Looking from the Operators' Perspective," operators are seeking ways to raise revenue per subscriber, and data services are one way to do this. Today, data services consist of more than just wireless web services; many Java ME applications rely on the network for their content. Requiring handset manufacturers to include Java ME on their devices leaves an open door for developers to create new applications that provide operators with new sources of revenue.

Providing Java ME on devices is more than just an operator requirement for many manufacturers. Some manufacturers, including Research In Motion (RIM), offer Java ME runtimes that both meet Java ME standards as well as include additional classes in their implementation, enabling developers to build novel applications atop the phone's fundamental platform. More frequently, however, you'll find the baseline Java ME implementation on a device. In either case, Java ME enables device manufacturers to build and bundle applications for their products more quickly than with existing embedded toolkits.

This is especially true for the growing number of dedicated devices that connect via home or municipal wired and wireless networks where the use of Java ME may not be a mandate. Java ME provides a ready alternative to closed, proprietary platforms for writing application software for wireless Internet devices, set-top boxes, and other embedded systems. Even when the end platform is closed to third-party developers, selecting Java ME can help device manufacturers bring their product to market by providing a more powerful and well-understood platform than an internally defined or purely embedded alternative.

Whether chosen because of a customer requirement, as an opportunity for differentiation, or to speed product development, Java ME provides important advantages over other platforms. Unlike its larger cousins, Java Platform, Standard Edition (Java SE) and Java Platform, Enterprise Edition (Java EE), Java ME has been carefully tuned to run on small devices, important for meeting the cost and power constraints of most devices today. It's an open platform, encouraging contributions of technologies through the Java Community Process (JCP). Finally, Java ME brings with it the entire community of Java developers, providing a pool of talented engineers, designers, and project managers from which to draw.

Looking from the Operators' Perspective

Wireless operators today face challenges, too. While differentiation on the basis of quality and brand remain important, chief among challenges is the drive for higher ARPU. Revenue from voice activity has largely leveled off, making data services an obvious area in

which to drive growth. This is especially true in some countries, including the United States, where carriers have spent huge sums of money obtaining the rights to new parts of the wireless spectrum.

While arguably the wireless Web, Short Message Service (SMS), and Multimedia Messaging Service (MMS) all play a role in contributing to the bottom line for data services, mobile applications play a growing role as well. Java ME applications can contribute in two ways: by driving data use itself and by providing revenue when an operator acts as the channel for application distribution. Increasingly, just as you can buy a ring tone or wallpaper for your handset via the operator's wireless web portal, you can now buy Java ME applications as well. As you will learn later in this chapter (in "Marketing and Selling Your Application"), partnerships between developers and operators establish important channels for application sales, bringing revenue to both parties.

While the bottom line drives business decisions, marketing and brand image play an increasing role in Java ME's importance for operators. By opening their network to third-party developers such as yourself, operators bring your creativity to the table. Moreover, Java ME enables operators additional ways to partner with key brands around the globe, helping the operators differentiate themselves and giving brands far removed from the mobile computing market an opportunity to interact with consumers in new ways.

Looking from the Consumers' Perspective

Today's consumers demand more from their devices. Whether using a cell phone, set-top box, or dedicated appliance, consumers expect clear value. Reliability, ease of use, personalization, and network awareness are features becoming increasingly important even for traditionally isolated devices. Java ME fits the bill as a platform on which to base these devices, because it's small, highly portable, and powerful.

In the wireless telecommunications market, consumer demand for reliability, ease of use, personalization, and network awareness has already begun and continues to grow. As I write this, Sun estimates that more than two *billion* wireless terminals have shipped with a Java runtime since the initial launch of Java for handsets. These devices support communication, entertainment, multimedia, and other applications bringing customization and choice to mobile device users around the world.

Java is poised to repeat this success within the set-top box market after years of persistent effort. With Java on every Blu-ray player as well as countless set-top boxes for personal entertainment, the potential for new applications is almost boundless. This market will be more diverse than the mobile market, with room for both small players and large application development houses as well as the traditional entertainment content partners (many of whom were latecomers to the wireless telecommunications marketplace).

Through all of this, Java ME enables developers to provide subscribers with greater choice, freedom, and flexibility. The fundamental platform enables developers to create stand-alone applications as well as network-aware applications and games, while

enabling manufacturers to add more interfaces to tap the custom features of the underlying platform.

Some pundits have accused the Java ME marketplace of "wagging the dog" to some extent—that is, driving consumer demand for applications for the sake of technology itself. While this claim is not wholly unwarranted, it's also the nature of a fast-paced market in which new products are tested on the market, and only those with successful business cases and clear value to consumers will survive. Java ME accelerates this process by providing an open standard on which to base the development of new products and services.

Looking Inside the Java ME Platform

The Java ME platform isn't really one platform, but rather a collection of platforms and libraries that work on a host of mobile devices. Even more confusing, Java ME began as a mobile environment for cell phones and personal digital assistants (PDAs), but has since expanded to include devices with similar constraints, including industrial devices, set-top boxes, Internet appliances, and other constrained platforms.

Justifying the Need for a Mobile Edition of Java

At first, the need for a mobile edition of Java may not be apparent. After all, today's cell phones are more powerful than the PCs that ran the first commercially available versions of Java more than a decade ago. However, a key feature of Java ME is its size and performance footprint. This is especially important for the constraints common to mobile and embedded devices. The constraints that mobility puts on size, power consumption, and cost mean that less capable processors and less memory will be found in devices for the foreseeable future. These constraints apply to less-mobile devices such as set-top boxes, too; in designing a commodity consumer device, every penny counts, so frequently low-cost (slower) processors and less memory are available. Moreover, as green manufacturing increasingly comes into play, fixed consumer devices will be subject to the same sorts of power constraints as mobile devices.

But Java ME isn't just about footprint. It's also about a new way of looking at computing. Many of the differences between Java SE and Java ME are about functionality, not footprint. The Java Application Descriptor (JAD) file is one example; it describes an application, including its name, icon, publisher, and other information. Or take Java ME's security model, which includes the notion of privileges and permissions for interfaces. Using Java ME, running applications may require individual privileges to perform sensitive operations, such as sending an SMS message, requesting the position of the handset via the location-based interface, or exchanging data using Bluetooth. Privileges are specified in the JAD file of a Java ME application and may be granted or denied depending on the origin of the application. (I discuss this in more detail in the "Packaging and Executing CLDC/MIDP Applications" section in Chapter 3.)

It's important to realize that the ultimate goal for Java ME is to provide an extensible yet highly portable, minimum-footprint, Java implementation that can run on a wide variety of network devices with constant or intermittent network connectivity. The platform emphasis is on application-level, not system-level, programming, and the application programming interfaces (APIs) that are supported reflect this distinction. Extensibility is another distinction, especially for one flavor of Java ME: the Connected Limited Device Configuration, which I discuss in the "Introducing the Connected Limited Device Configuration" section later in this chapter. Extensibility is a key differentiator between the platform, other Java platforms, and other computing platforms as a whole.

Making Java Work on Mobile Devices

Chapter 2 looks at the changes made to Java and its base classes for Java ME in close detail, but it's worth summarizing these changes now:

- The Java runtime must have the ability to reject invalid Java class files to ensure system security and integrity.
- The Java runtime controls an application's access to specific parts of the system (such as the file system, network access, and so forth).
- Applications run within a sandbox that prevents unauthorized access to other applications and libraries.
- The environment must support the ability to download new applications, but cannot use this mechanism to modify or override protected system classes in any way (including changing the order in which classes are looked up).
- The platform libraries may lack specific interfaces for performance and memory use reasons; for example, one configuration of Java ME doesn't support object finalization, nor does it have support for the Java Abstract Window Toolkit (AWT) or Swing user-interface libraries.
- The platform may or may not have support for floating-point mathematical operations, depending on the version.

For brevity, I've painted this list with a broad brush; not all of these changes apply to all flavors of Java in the Java ME family, as you'll learn in the next section and Chapter 2.

Because of the immense variety in devices supported by Java ME, there are actually different implementations of Java for different devices. Specifically, how Java ME functionality is defined for a specific device is based on three concepts:

- *Configuration*: Defines the basic set of libraries and Java virtual machine (VM) for a broad range of devices
- *Profile*: Defines a set of APIs for a narrower range of devices
- *Package*: A set of optional APIs that pertain to a specific technology, such as multimedia or Bluetooth access

Figure 1-1 shows how these abstractions stack to define the software characteristics of a device.

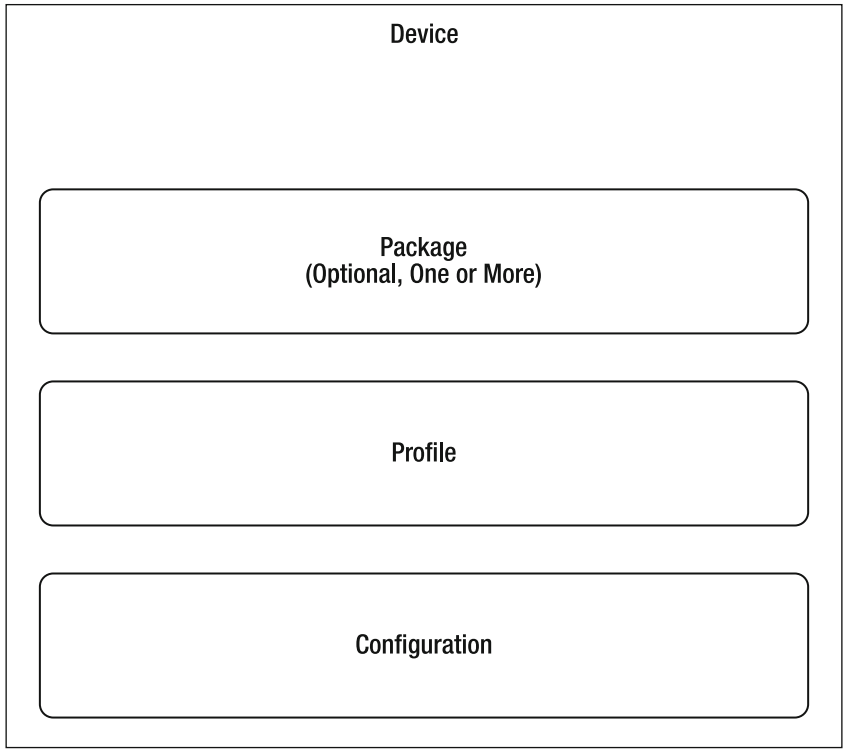


Figure 1-1. *The relationship between configurations, profiles, and packages in Java ME*

Each instance of a configuration, profile, or package *should* have as its basis one or more Java Specification Requests (JSRs) that document the purpose and interface for the Java extension in question. Appendix A summarizes the JSRs relevant to Java ME that define functionality discussed in this book.

Today, there are two configurations within Java ME: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). There are a handful of profiles that sit atop either of these configurations, and many, many packages. (In the next section, I explore these configurations in greater detail.)

The most commonly known tuple of configuration, profile, and package is based on the CLDC package, including the Mobile Information Device Profile (MIDP) accompanied by optional packages, powering billions of the world's mobile phones today. In fact, this configuration is *so* common that many think of it as the only version of Java ME. Figure 1-2 shows a typical mobile-phone configuration with a few common optional packages. I discuss the MIDP in depth in the "Introducing the Mobile Information Device Profile" section later in this chapter.

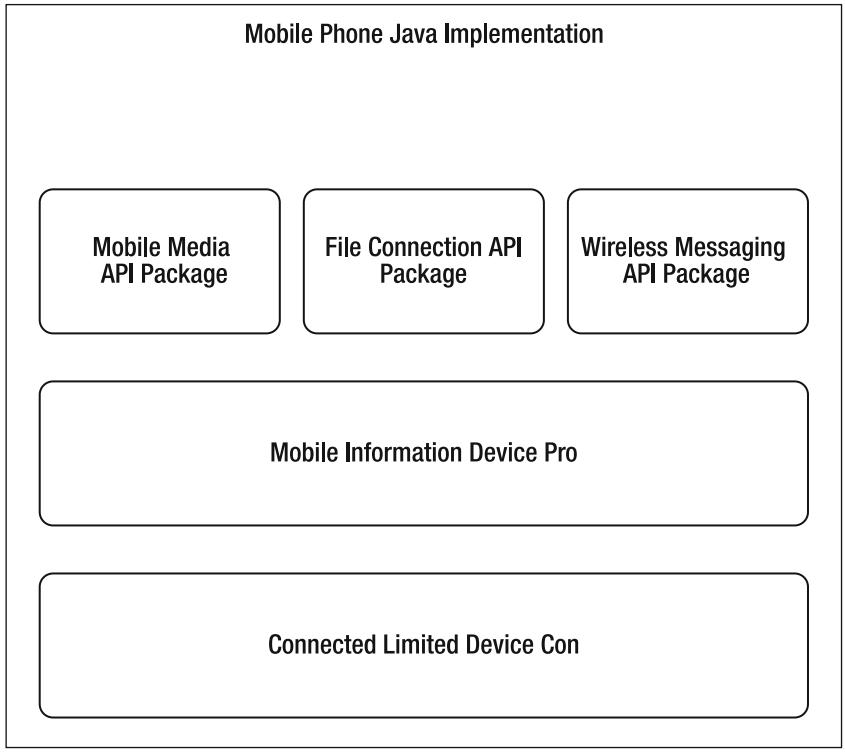

Mobile Phone Java Implementation
Mobile Media API Package
File Connection API Package
Wireless Messaging API Package
Mobile Information Device Pro
Connected Limited Device Con

Figure 1-2. *The relationship between configurations, profiles, and a few of the packages in a typical cell phone*

Another, increasingly common tuple is based on the CDC package and powers set-top boxes and other devices. Such a digital media configuration is for higher-end connected devices, and it incorporates the Foundation Profile as well as the Personal Profile, and perhaps additional optional packages. Unlike the CLDC-MIDP configuration, this configuration defines a subset of the Java AWT, bringing a well-understood graphics library to mobile devices. Figure 1-3 shows a typical configuration based on the CDC for today's consumer electronics devices.

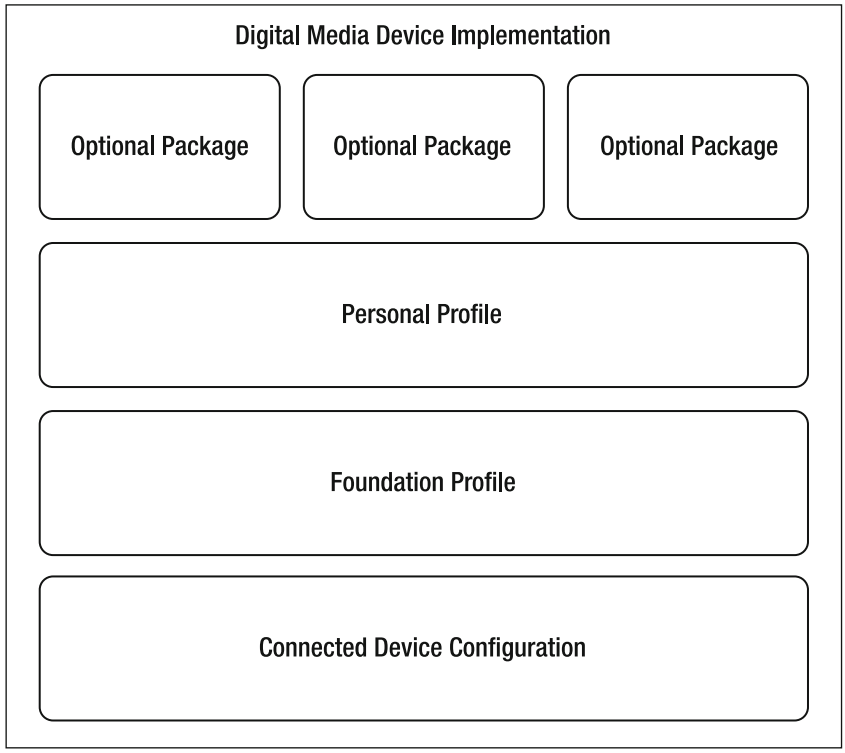


Figure 1-3. *The relationship between configurations, profiles, and packages in a typical set-top box or other digial media hardware*

Understanding Configurations

The fundamental building block of the Java ME architecture is the configuration. A configuration defines the Java Virtual Machine (JVM), basic language support, and fundamental classes for the widest possible set of devices, such as all mobile wireless terminals or all set top box–like devices. A specific device must meet the requirements of at least one configuration, such as the CLDC, in order to qualify as a Java ME device.

Introducing the Connected Limited Device Configuration

The CLDC is explicitly designed for memory-constrained devices that may be always or intermittently connected to the network. It defines a Java environment based on the Java K virtual machine (KVM), a modified version of the JVM that has been specially tuned to operate on low-power devices with a 16- or 32-bit processor and as little as 192KB of RAM. The most common deployment of the CLDC, as I've already noted, is in the billions of Java-enabled cell phones that have shipped in the last several years.

A device with the CLDC is quite primitive; it's missing many of the classes you'd expect to find that permit application development (such as user-interface widgets, network connectivity, and the like). Instead, the implementation leaves those details to profiles atop the CLDC, and focuses instead on reducing memory and CPU footprint to the absolute minimum necessary for the Java environment. One area obviously affected is the nature of the libraries included in the CLDC; many Java SE classes are simply not available in the CLDC, and of those that are available, not all methods may be available. This is especially true for collections, where only three classes and one interface are available. (For more details on exactly what *is* supported, see Chapter 2.)

While the JVM historically absorbs the entire burden of Java bytecode verification, this is not the case for the CLDC. Because bytecode verification is expensive both in terms of processor and memory, the responsibility of bytecode verification is shared between the developer and the KVM running on the mobile device. As part of the build process, you run a tool called preverify that inlines subroutines in each class file (removing certain bytecodes in the process) and adds information about variables on the stack and their types. At runtime, the KVM uses this additional information to complete the bytecode verification prior to execution.

■ **Note** Two-pass preverification obviously brings with it potential security issues, because a malicious developer could inject code that appears to be preverified but does not meet all of the standards required by the CLDC. To address this, CLDC applications are typically downloaded from trusted sources; moreover, the profile used with the CLDC—the MIDP—adds code signing, so that a Java implementation can verify the originator of the code being executed and provide an appropriate level of trust.

But security changes don't end there. The sandbox model, familiar to applet developers from the early days of Java, plays a much greater role in the CLDC, where applications are sandboxed from each other as well as the host operating system. Each application runs in an environment where the only facilities it can access are those classes in its own distribution and the local classes provided by the device's configuration, profile, and optional packages. The download, installation, and management of other Java applications are also inaccessible, preventing one application from affecting another. Finally, there is no facility for creating native methods, closing potential holes between the sandbox and the native platform.

Another key feature of the CLDC is the Generic Connection Framework (GCF), which defines a hierarchy of interfaces that generalize port connections. The resulting hierarchy provides a single set of classes you use to connect to any network or remote port, including Transmission Control Protocol over IP (TCP/IP), User Datagram Protocol (UDP), and serial ports, just to name a few. The GCF is defined in the `javax.microedition.io` package.

Due to processor and power constraints, CLDC 1.0 did not support floating-point mathematics; CLDC 1.0 devices could only perform integer math. This changed with the advent of CLDC 1.1, as CLDC 1.1 devices must support floating-point operations. However, as a developer, you should be aware that floating-point mathematics remains computationally expensive. Be careful when choosing to use them, as they can cause performance issues within your applications.

Introducing the Connected Device Configuration

The CDC is for devices that are more capable than those used by the CLDC, such as high-end PDAs, set-top boxes, and other Internet appliances. As such, the goals of the CDC are slightly different than those of the CLDC. Instead of targeting the largest possible number of low-cost hardware, the CDC focuses on leveraging developer and technology skills from the existing Java SE platform while respecting the needs of resource-constrained devices.

Unlike the CLDC, the CDC virtual machine meets the same requirements as the JVM that powers Java SE. In fact, if you add to the CDC the profile and packages usually found on a media-capable device, you'll find little that distinguishes the environment from a Java SE platform except the *extra* APIs that additional packages may provide. This is the strength of Java, and especially the more robust CDC: you can leverage your Java skills across the entire product family line. In addition, the CDC includes all of the Java language APIs required of the CLDC, including the GCF.

Packages containing classes defined by the CDC include `java.lang`, `java.io`, `java.net`, `java.security`, `java.text`, and `java.util`. Devices running the CDC must also support CLDC APIs and packages; this enables the fullest possible support for all Java applications. While such devices are rare on the market today, it's an obvious future direction for Java, as the majority of devices continue to become more and more powerful.

Understanding Profiles

Profiles collect essential APIs for building the most fundamental of applications across a family of devices. The most well known profile by far is the MIDP, which powers mobile phones and sits atop the CLDC. Equally important is the Foundation Profile, which is analogous to the MIDP for providing application support classes. Unlike CLDC-based devices, however, CDC-based devices typically also include either the Personal Basis Profile or the Personal Profile, or both, to provide user-interface support for applications.

Introducing the Mobile Information Device Profile

The MIDP is the foundation of today's mobile-phone Java revolution (Part 2 of this book is dedicated to the MIDP). Defining a number of new interfaces, it enables you to develop *MIDlets*, which are applications that run atop the CLDC. The MIDP defines these other interfaces, as well:

- The application life cycle via the MIDlet
- Networking
- Persistent storage
- Sound
- Timers
- The user interface (including the portable display and input as well as support for games)

However, the MIDP defines more than just interfaces and the classes that support those interfaces. It also describes how applications are installed on a device. While the actual implementation may differ from device to device, the general requirement is that from the device you're able to browse applications, select one for download using HTTP/1.1, and have the device install the MIDP and present it in its application manager. Applications are accompanied by an application descriptor (see the "Packaging and Executing CLDC/MIDP Applications" section in Chapter 3) that includes information about the application, such as the application vendor, application name, and application size.

The MIDP defines the notion of *permissions*, which indicate that a MIDlet can access a restricted API. The only permission defined by the MIDP pertains to network connections, but packages are free to introduce other permissions. A permission is a name using the same prefix and class or interface name as the name of the restricted API. For example, to use a network socket, a MIDlet needs the permission `javax.microedition.io.Connector.socket`. This permission accompanies the application in the descriptor file, and a MIDlet can test for the presence of a privilege using the MIDlet method `checkPermission`, which returns `1` if the permission is granted, `0` if it is not, and `-1` if the result is indeterminate, such as a requirement that the user be asked to grant permission manually.

The MIDP also defines the notion of a *trusted application*, which is permitted to use a restricted API, such as the file connection package, to access the file system. An application gains trust by virtue of the domain from which it came; typically the application carries this information through a cryptographic signature applied by a certification authority or carrier. As a result, even when bearing privilege, applications must be prepared for security exceptions, which may occur when an application is untrusted and attempts to access a restricted API.

Introducing the Foundation Profile

The Foundation Profile, targeted for CDC-enabled devices, runs on devices with less than 256KB of ROM (of course, a Foundation Profile device can have more ROM, but that's the minimum supported by the profile), a minimum of 512KB of RAM, and a persistent network connection. In many ways, the Foundation Profile is far less ambitious than the MIDP. In conjunction with the CDC, it provides network and input/output (I/O) support, but no classes for application development; those are relegated instead to the Personal Basis Profile and the Personal Profile.

Classes augmented by the Foundation Profile include those in the `java.lang`, `java.io`, `java.net`, `java.security`, `java.text`, and `java.util` packages. These classes are based on classes found in the Java SE 1.4 class hierarchy, as well as additional `javax.microedition.io` classes that provide HTTP and HTTP-over-Transport Layer Security (TLS) network operations.

Introducing the Personal Basis Profile

Most CDC-based devices have at least some user-interface requirements. The most highly embedded devices may use only the Foundation Profile with a custom package atop that to provide support for a custom-made liquid-crystal or light-emitting diode (LED) display, but by far the most common are devices with raster displays that need rich graphical user interfaces (GUIs). To accommodate this in a standard way, two profiles are available. The smaller of the two, the Personal Basis Profile, actually provides *two* class hierarchies for applications: the applet model, and a new hierarchy for media devices that defines the Xlet programming model. Xlets are similar to applets, except they have a life cycle that supports being paused and resumed, which is important for media devices in which multiple applications and media streams may interrupt an application's execution at any time.

The Personal Basis Profile also defines a subset of the AWT for GUI development. Unlike the traditional AWT, the Personal Basis Profile defines a *lightweight* control facility, in which user-interface components draw themselves rather than have peer controls derived from the native platform. (The Java Swing implementation takes the same approach of having the Java environment draw its own controls.) Consequently, the Personal Basis Profile only includes support for `java.awt.Window` and `java.awt.Frame` (which hook to the native platform's windowing manager and contain lightweight components), and `java.awt.Component` and `java.awt.Container` (which are lightweight components used to create all of the other components in the hierarchy). Note that the Personal Basis profile does not define traditional AWT controls, including buttons, lists, and other items, because these would have connections to peer components from the native platform. Instead, you can create your own components or import a package on a specific platform that provides the components you need.

Finally, the Personal Basis Profile also includes classes that support communication between Xlets, using a subset of Java's Remote Method Invocation (RMI) API. It's important to remember, though, that while parts of the `java.rmi` package are included in the Personal

Basis Profile, the profile does *not* support RMI; just enough of the RMI implementation is included to facilitate communication between two Xlets running on the same device.

Introducing the Personal Profile

The Personal Profile is a superset of the Personal Basis Profile that provides support for the entire AWT, as well as limited JavaBean support. Some readers may remember PersonalJava, the predecessor to Java ME that was targeted for higher-end Internet appliances and set-top boxes; the Personal Profile atop the CDC is the forward migration path for applications running on PersonalJava.

In fact, the Personal Profile is almost the same as Java SE 1.4.2, with these differences:

- Support for RMI is available through an optional package (`java.rmi`).
- Support for SQL is available through an optional package (`java.sql`).
- Support for Java Swing is available through an optional package (`java.swing`).
- Support for Object Management Group (OMG) interfaces, including Common Object Request Broker Architecture (CORBA), is available through an optional package (`org.omg`).
- There is no support at present for the Java Accessibility API (`javax.accessibility`).
- There is no support at present for the Java Naming and Directory Interface (JNDI) in `java.naming`.
- There is no support for the Java Sound API (`java.sound`).
- Support for JavaBeans is limited to runtime support; there is no support for bean editors running directly on a CDC environment.
- The applet API `getAccessibleContext` is not supported.
- Applet and AWT methods deprecated from Java SE have been removed from all supported classes.

Understanding Packages

A package, as its name implies, is an object or group of objects in a common container. As important as platforms and profiles are to the modularity of the Java ME platform, packages are arguably the key to Java ME's continued success, as they permit Sun and third-party vendors to extend the Java ME platform for specific families of devices.

There are countless packages for Java ME; many of the now-standard interfaces that are part of successful MIDP-based devices are in fact packages. In this book, you will learn how to use several packages, including

- The GCF, documented in JSR 30
- The `FileConnect` interface, which provides local file access on MIDP and is documented in JSR 75
- The Java Bluetooth API, documented in JSR 82
- The Wireless Messaging API, documented in JSR 120
- The Web Services API, documented in JSR 172
- The Java Advanced Graphics and User Interface (AGUI) API, for CDC devices, documented in JSR 209
- The Java Mobile Service Architecture (MSA), documented in JSR 248

Planning Your Approach to Java ME Development

Java ME's strength is rooted in the ubiquity of Java today. However, with this ubiquity comes challenges. The multitude of APIs and the diversity of distribution channels make planning the technical and business aspects of your application equally important.

Selecting Appropriate Device Targets

As you've seen, the triad of configurations, profiles, and packages means managing a *lot* of different APIs. For many kinds of applications, this may not be a serious problem—most productivity and network applications need just a network layer, some GUI elements, and a persistent store, which is available under just about any combination of configuration and profile you might encounter.

That's not always the case, however. Your application might depend on functionality present in only a specific package, perhaps, either by design (say, a Bluetooth-derived application for proximity detection) or product differentiation. There's always the temptation of using an optional package to speed time to market, too, only to find later that it's not available on the next target for your product.

Consequently, if portability is important, you should base your application on as few Java ME packages as you possibly can. Obviously, this doesn't mean creating your own control framework from scratch or implementing your own web services framework from scratch if you don't have to. But it does mean understanding what APIs are

available on the devices you're targeting, and performing regular surveys of the market. I find it best to keep track of what APIs I plan to use as I design and implement my application, and correlate them against the JSRs that define those APIs. Then, when it's time to bring my application to market on a different device, I can check to see which packages are offered by the new hardware and scope my porting effort accordingly. Sites like the Wireless Universal Resource File (WURFL) device description repository at `http://wurfl.sourceforge.net` and the support database for J2ME Polish (a library originally targeted at Java ME's predecessor, J2ME, that simplifies cross-device development) at `http://devices.j2mepolish.org/` are invaluable in planning your product launch or porting efforts. A little research as you design your application can pay big dividends when rolling it out to consumers.

Marketing and Selling Your Application

There are a myriad of channels for Java ME applications today, each with their own set of business challenges. Many readers see wireless operators as the logical channel for application distribution, given the popularity of the MIDP today. Still others target web distribution to hardware directly or bundle applications with hardware at manufacturing time. A small percentage of you may be working directly with platform or hardware manufacturers, and so your channel to the consumer (and the notion of a consumer itself!) is quite different.

You can distribute your application to consumers in a number of ways. Certainly direct distribution is a possibility, by publishing a link to your application (see Chapter 3 for how to package your application for the different configurations). This may sound simple, but it poses an obvious business question: How will you get paid for your application? Free distribution, advertising, and per-download or subscriptions via credit card or PayPal fulfillment are all possibilities.

Because of the need for privilege by most applications and revenue by most developers, a typical deployment for a mobile application involves both third-party certification and business negotiations. The process of third-party certification typically involves a business program such as Java Verified, which tests your application, and assuming it passes testing, cryptographically signs your application for distribution. Usually accompanying this signed endorsement is access to additional privileges; for example, most MIDP implementations won't permit HTTP transactions without prior user approval unless a member of the Java Verified testing authority has signed the application. Once signed, you can distribute your application through aggregators, who broker the carrier relationship on behalf of you and many other developers, reimbursing you for sales (typically via a premium-SMS push or direct billing). Another distribution path is to negotiate with one or more wireless operators to distribute your application. This involves crafting the business relationship—how much will consumers pay for your application, and how will it be obtained?—as well as additional testing, which usually results in a second cryptographic signature for your application.

With this signature may come additional privileges, such as access to location-based APIs. Negotiating operator partnerships can be an expensive and time-consuming task, but without the help of an intermediate aggregator, you need to perform these negotiations for each operator's network on which you want to deliver your application.

This issue of privileges and cryptographic signatures isn't just a business issue, but can be a functional issue as well. This may very well affect the functional requirements for your application; for example, Acme Wireless (a fictitious operator) might only permit applications access to location-based interfaces for applications *it* signs and distributes. If your application needs positioning data (say, to locate the user to recommend restaurants or services), that requirement won't be just a technical requirement but a business requirement to establish the necessary operator relationship to ensure the required privilege. This admittedly adds risk to your business model, because it's difficult to ascertain in advance whether or not specific operators will carry your application, and obtaining this information in advance can involve significant business investment. It does, however, give you certain assurances if you can bridge the gap and obtain operator signing and distribution for your application, because this gives significant placement and marketing clout for your application. While all of these comments apply primarily to Java ME MIDP applications, due to the large number of MIDP applications and MIDP-capable devices on the market, I suspect that the landscape for Java CDC applications will be similar, given the rampant success of the MIDP distribution model.

Of course, if you're a developer involved in planning your business strategy, don't forget other avenues for distribution, too. Direct-to-manufacturer deals, while difficult to obtain, are potentially lucrative sources of long-term revenue. Moreover, providing services for Java consulting remains an important mainstay for many developers. Thinking creatively, a host of business models can support your application's development and distribution.

Wrapping Up

Java ME meets needs for operators, device manufacturers, and consumers. By providing a rich set of platforms across devices from low-end mobile phones to high-end set-top boxes, and from embedded devices to high-end PDAs, Java ME enables software developers like you to leverage your skills to bring new applications to market.

The Java ME platform is divided up into configurations, platforms, and profiles. Two configurations are presently available: the CLDC, which is targeted for constrained devices, and the CDC, which is targeted for more robust devices. Atop these configurations are one or more profiles, such as MIDP for the CLDC, or the Foundation, Personal Basis, and Personal Profiles for the CDC. These provide additional APIs in a standard way that let you write your application for a family of devices. Finally, packages allow Sun and third parties to extend the APIs on a device or suite of devices in a standard way, giving access to new functionality in portable ways.

CHAPTER 2

Shrinking Java to Fit

Making Java run on constrained devices remains a challenge, even nearly a decade after the first attempts to do so. However, the configurations provided by Java ME are helping to meet this challenge by bringing the Java platform to the widest possible selection of devices. By splitting Java into separate configurations—the CLDC for devices with the lowest possible throughput, and the CDC for constrained mobile devices with moderate memory and processing power—Java ME can provide a computing environment for nearly every mobile device.

In this chapter, I show you the explicit differences between Java ME and Java SE. I begin with the CLDC, showing you just what's different between past and present versions of the CLDC's virtual machine and the virtual machine running the Java SE. Next I look at the CDC the same way. Finally, I turn your attention to the class libraries accompanying each of these configurations, showing you precisely which classes both the CLDC and the CDC support.

Making It Fit: The CLDC

To understand the limitations of the Java ME CLDC, it's important to understand a bit about the *history* of the CLDC itself. The CLDC stems from early in the history of Java when mobile devices were veritable cripples by today's standards. Mobile phones had well under a megabyte of heap and were powered by processors that would be considered anemic by today's standards. The initial release of the CLDC was based on an entirely new virtual machine for small devices, the Java KVM. Consequently, it lacked a lot of what Java developers were used to. CLDC 1.0, which was documented in JSR 30 in 2000, was the starting point for what has become the Java ME platform.

To keep pace with developments in hardware capabilities and manufacturing costs, an expert working group of more than 20 companies revised the CLDC standard, resulting in JSR 139, which is the standard that defines CLDC 1.1. This standard, completed in 2002, took into account the growing capabilities and markets for mobile devices, and it added back some features stripped from Java to make Java fit on mobile devices during the brief period that CLDC 1.0 was available.

The distinction between the CLDC 1.0 and 1.1 releases may or may not affect your development approach, as the overwhelming majority of new devices sold today support CLDC 1.1 (or the CDC, in fact, for some higher-end devices). For example, as I write this, virtually all wireless operators' Java-enabled *feature phones*—those phones subsidized by operators to provide a mix of features at a low cost to drive both voice and data uses—are CLDC 1.1–compliant. Even accounting for device churn (the rate at which users buy new devices), there are countless CLDC 1.0 devices on the market, and a savvy developer may well be able to wring revenue from these users. On the other hand, many of the compelling features of the Java ME platform (available through packages documented by the JSR process, as I discuss in Chapter 1), such as multimedia and 3D graphics, are only available on relatively new handsets, which typically run CLDC 1.1 and MIDP 2.0 or 2.1. As a result, when you plan your application, you should balance the need for APIs provided by a later version of the CLDC or MIDP with the business case provided. You'll need to answer this question: do the technology requirements for your application so drastically reduce the number of devices on which you can deliver your application that your business case becomes invalid? This is a common theme for cutting-edge Java ME developers, but less so now than in years past, and over time (as I discuss in the section "Looking Toward the Future of the CLDC" in this chapter) it should diminish entirely.

Understanding the Present: CLDC 1.1

Cost and power consumption significantly limit the capabilities of even today's mobile devices. While consumer demand for high-capability personalizable devices remains high, manufacturing cost remains a key factor and drives the selection of low-cost components (meaning less memory, slower processors, and so forth). While Moore's law has driven ever-faster and ever-cheaper devices in the hands of consumers, power sources have been unable to keep up, making power management another key challenge for mobile devices. Consequently, there continues to be constrained devices and a role for the CLDC on those devices.

Specifically, CLDC 1.1 differs from Java SE in the following ways:

- *CLDC 1.1 offers no finalization of object instances*: CLDC libraries, by definition, do not include the method `Object.finalize`.

- *CLDC 1.1 doesn't support asynchronous exceptions*: The virtual machine does not throw asynchronous exceptions, and you cannot invoke `Thread.stop`, as it doesn't exist in the CLDC.

- *CLDC 1.1 supports only a limited number of the* `java.lang.Error` *subclasses*: For other errors, the CLDC either halts the virtual machine in an implementation-specific manner or throws the nearest defined `java.lang.Error` subclass. (See the "Changes to java.lang.Exception" section later in this chapter.)

- *CLDC 1.1 doesn't support thread groups or daemon threads*: The CLDC does support multithreading, but if you need to manage a group of threads, you must use individual thread operations on members of a collection.

- *Class file verification is a two-step process in CLDC 1.1*: As I discuss in the "Building CLDC/MIDP Applications" section in Chapter 3, the CLDC requires a preverification step during application development and a subsequent completion of verification after downloading.

- *For security reasons, CLDC 1.1 doesn't support user-defined class loaders*: The CLDC must have an internal class loader that other code cannot override, replace, or reconfigure in any way.

With the possible exception of CLDC 1.1 lacking class finalization and thread groups, these differences are unlikely to affect your approach to application development. Larger in scope, however, is the reduction of the CLDC class library itself, which I explore later in the chapter (in the section "Changing the Java Class Library to Fit the CLDC").

MOORE'S LAW AND MOBILE DEVICES

Moore's law—the observation that the number of transistors that can be fit on the die of an integrated circuit is increasing exponentially—has driven many facets of the computing revolution, including the increases in computing speed and memory density for more than 50 years. Documented by Gordon Moore in an article in *Electronics Magazine* in 1965, the law isn't one of nature, but rather a recognition of the economies of scale provided by integrated circuit production.

For mobile devices, Moore's law brings two key points: the ever-dropping cost of each transistor means that more computing power can be purchased for less money, and that the increased transistor density brings faster devices with more memory in portable form factors at the same exponential pace as it brings to desktop users. That is not to say, however, that the capabilities of mobile and desktop devices will *converge*, as Moore's law is in play for larger computing platforms as well.

Moreover, one area remaining largely untouched by Moore's law is in providing energy for mobile devices. While the battery industry has seen many improvements in technology over the last two decades, these advancements have not been exponential in nature, leading to aggressive work on the part of hardware and software developers to manage power consumption at the hardware and software levels. This has a direct impact on computational resources, of course, which consume power while operating.

These factors are likely to persist into the future, meaning that while mobile devices continue to get increasingly powerful, it's unlikely that mobile devices and desktops will ever truly converge. Instead, it's likely that mobile devices will become sufficient to support the majority of operations performed by yesterday's desktops—something I talk more about in the "Looking Toward the Future of the CLDC" section later in this chapter.

Looking Back at CLDC 1.0

CLDC 1.0 had significantly more limitations than CLDC 1.1, as you can imagine when you consider the time in which it was developed. When developing a CLDC 1.0–compliant application, you must consider the following limitations:

- *No floating-point operations*: CLDC 1.0–compliant virtual machines have no support for floating-point byte codes, nor do they have library APIs that perform floating-point math. These limitations mirror the capabilities of the processors typically targeted by CLDC 1.0.
- *No support for weak references*: The `WeakReference` declaration is not supported by CLDC 1.0–compliant virtual machines.
- *No support for named threads*: The CLDC-compliant virtual machines have no support for naming threads.
- *Different uses of* `java.util.Calendar`, `java.util.Date`, *and* `java.util.TimeZone` *classes*: These classes differ significantly from their Java SE counterparts.

Looking Toward the Future of the CLDC

Although I believe that true convergence of capability between mobile devices and fixed devices remains a long way off (see the "Moore's Law and Mobile Devices" sidebar), mobile devices are arguably reaching parity with desktop devices in terms of basic usability for many applications, including data processing, multimedia, entertainment, social networking, and user-generated content. Already, some high-end wireless terminals, such as those sold by Nokia and Sony Ericsson, offer CDC-compliant runtimes as well as CLDC-compliant runtimes. Looking into the future, representatives of Sun have already stated that eventually the Java ME and Java SE interfaces will merge into a single platform.

Does this make the CLDC less relevant? Certainly not, as literally billions of devices running the CLDC are already in consumers' hands, and billions more are to be shipped in the coming years. Instead, expect to see a filling in of the gaps between the CLDC and the full Java SE platform, much as the transition from CLDC 1.0 to CLDC 1.1 filled in gaps.

Making It Fit: The CDC

In some ways, the CDC has had a more tumultuous history than its smaller cousin, the CLDC. Beginning life as PersonalJava and based on Java 1.1.8, the CDC is actually quite old, dating back to the late nineties. PersonalJava consisted of a heavily optimized Java VM and an associated class library that incorporated most of the Java stack, including a GUI, for application developers on set-top boxes, high-end connected wireless terminals, and other products.

A few years ago, Sun announced the trio of flavors for Java: Java 2 Platform, Micro Edition (J2ME), Java 2 Platform, Standard Edition (J2SE), and Java 2 Platform, Enterprise Edition (J2EE). While PersonalJava remained, it was subsumed into Java ME and later reached its end of life, meaning that Sun withdrew support for the platform.

At the same time, Sun introduced JSR 36, defining the CDC. The CDC targets devices requiring an entire virtual machine for hardware that, through optional extensions, may run the entire Java 2 platform. JSR 36 bases its implementation on Java SE 1.3, and it includes all additional interfaces defined by the CLDC, providing a migration path forward from CLDC devices. JSR 218, defining CLDC 1.1.2, normalized the interfaces provided against Java SE 1.4.2, helping ensure parity going forward between the CDC and larger Java environments.

It does little good to identify each and every class in the CDC, because the list is the same as the list of classes for Java SE. It is, however, worth your while for me to call out the *packages* included in the CDC, because while the CDC provides a baseline for compatibility, it does not provide the user-interface classes necessary to build a full application. Table 2-1 shows the packages provided by the CDC.

Table 2-1. *Packages Provided by CDC 1.1.2*

`java.io`
`java.lang`
`java.lang.ref`
`java.lang.reflect`
`java.math`
`java.net`
`java.security`
`java.security.cert`
`java.text`
`java.util`
`java.util.jar`
`java.util.zip`
`javax.microedition.io`

The CDC is a more restrictive environment than Java SE. Be aware of the following:

- If the CDC implementation with which you're working supports invoking native methods, you must use Java Native Interface (JNI) 1.1 to interface with those native methods.
- If the CDC implementation with which you're working supports a debugging interface, you must use the Java Virtual Machine Debug Interface (JVMDI) for debugging.
- The CDC doesn't support secure code signing, certificates, the key store, and the JDK 1.1 `java.security.Identity` and `java.security.IdentityScope` interfaces.

Typically, any CDC implementation you encounter is accompanied by one or more profiles, such as the Foundation Profile, giving you support for GUIs and other capabilities.

Changing the Java Class Library to Fit the CLDC

Shrinking the Java virtual machine was only part of the trick to making the CLDC fit on mobile devices. One of Java's key benefits to developers is its robust class library. However, all of that code is also in Java and consumes precious read-only memory space on constrained devices, and some classes may not even be appropriate because of their runtime memory consumption. Of course, without its class hierarchy, Java wouldn't be nearly as compelling a platform, so the architects of the CLDC struck a balance, establishing an environment with relatively few Java interfaces that can run on the widest possible variety of devices while supporting a great number of applications. (And of course, device manufacturers are free to add—and have added—a great number of additional interfaces.)

Besides the obvious—as you know, neither the AWT nor Java Swing are part of the CLDC—many other classes, and some methods of other classes, are omitted from the CLDC. In the following sections, I show you which parts of the `java.lang`, `java.util`, and `java.io` hierarchies are different between CLDC 1.0, CLDC 1.1, and Java SE, as well as the additions made by the CLDC to the Java family of libraries.

Changes to the java.lang Package

Table 2-2 lists the classes inherited from the `java.lang` package. Many of these classes are a subset of the implementation found in Java SE.

Table 2-2. *java.lang Classes Supported by CLDC Version*

CLDC 1.0	CLDC 1.1
ArithmeticException	ArithmeticException
ArrayIndexOutOfBoundsException	ArrayIndexOutOfBoundsException
ArrayStoreException	ArrayStoreException
Boolean	Boolean
Byte	Byte
Character	Character
Class	Class
ClassCastException	ClassCastException
ClassNotFoundException	ClassNotFoundException
Error	Error
Exception	Exception
IllegalAccessException	IllegalAccessException
IllegalArgumentException	IllegalArgumentException
IllegalMonitorStateException	IllegalMonitorStateException
IllegalThreadStateException	IllegalThreadStateException
IndexOutOfBoundsException	IndexOutOfBoundsException
InstantiationException	InstantiationException
InterruptedException	InterruptedException
Integer	Integer
Long	Long
Math	Math
NegativeArraySizeException	NegativeArraySizeException
NumberFormatException	NumberFormatException
NullPointerException	NullPointerException
Object	Object
OutOfMemoryError	OutOfMemoryError
Runnable	Runnable (interface)
Runtime	Runtime
RuntimeException	RuntimeException
SecurityException	SecurityException
Short	Short

Continued

Table 2-2. *Continued*

CLDC 1.0	CLDC 1.1
String	String
StringBuffer	StringBuffer
StringIndexOutOfBoundsException	StringIndexOutOfBoundsException
System	System
Thread	Thread
Throwable	Throwable
VirtualMachineError	VirtualMachineError
	Double
	Float
	NoClassDefFoundError
	Reference
	WeakReference

Changes to java.lang.Exception

As I stated previously, the exception hierarchy for the `java.lang` package is significantly smaller than the hierarchy for Java SE.

Changes to java.lang.Object

The key change to the `Object` class—and therefore the *entire* Java hierarchy—is the absence of the `finalize` method. As you know, the memory manager invokes a method's `finalize` method just before garbage collection; in the CLDC, there's no way to hook the garbage-collection process in this manner. This is as it should be: memory is often a precious commodity on Java ME devices anyway, so you should explicitly clean up resources when you're finished using them.

The Reflection API is also not part of the CLDC; the memory footprint it demands just isn't feasible. Of course, without reflection, there's no RMI, and without RMI, there's no Jini—Sun's platform for Java-based distributed computing.

Changes to java.lang.Math

As you might imagine, with the removal of floating-point support in CLDC 1.0, the `java.math` class was stripped of floating-point operations, leaving only operations with all-integer signatures such as `min` and `max`. With the reintroduction of floating-point

support in CLDC 1.1, the library was again brought to parity with Java SE, and these methods returned to the library.

Changes to Multithreading

The core essence of multithreading remains unchanged between the CDLC and the implementations of Java for larger-footprint devices. However, some key differences exist:

- CLDC 1.0 doesn't support thread naming and the `interrupt` method; however, you can find them in CLDC 1.1.
- The `suspend`, `resume`, and `stop` methods, now deprecated in Java SE, are wholly absent from the CLDC.
- The CLDC doesn't support thread groups and daemon threads.

Changes to java.lang.Runtime

The runtime facility is vastly changed between Java SE and Java ME for both security and memory reasons. It provides five methods:

- `getRuntime`: This static method returns the global runtime.
- `exit`: Although you can invoke this method, it won't exit the runtime; the entire application life cycle is managed by the application manager and MIDlets (see Chapter 3 for more details).
- `totalMemory`: You can use this method to determine the amount of total memory that's available.
- `freeMemory`: You can use this method to determine the amount of free memory that's available.
- `gc`: You use this method to invoke a garbage-collection operation.

Changes to java.lang.System

The `System` class has been heavily modified as well. There's no input stream, because CLDC devices don't have a traditional console (in fact, there is no way to obtain information written to `System.out` and `System.err` on most devices, either). The `System` class also provides a property accessor called `getProperty` to obtain information such as the name of the device host, the character-encoding scheme used by the device, the CLDC platform version, the MIDP version, and so forth.

Changes to java.lang.String and java.lang.StringBuffer

The key difference between these classes in Java SE and the CLDC is that CLDC 1.0 omits the methods that take floating-point arguments.

Changes to the java.util Package

Table 2-3 lists the supported classes in the `java.util` package.

Table 2-3. *java.util Classes Supported by CLDC Version*

CLDC 1.0	CLDC 1.1
`Calendar`	`Calendar`
`Date`	`Date`
`EmptyStackException`	`EmptyStackException`
`Enumeration`	`Enumeration` (interface)
`Hashtable`	`Hashtable`
`NoSuchElementException`	`NoSuchElementException`
`Random`	`Random`
`Stack`	`Stack`
`Timer`	`Timer`
`TimerTask`	`TimerTask`
`TimeZone`	`TimeZone`
`Vector`	`Vector`

Changes to Collections

The Java SE Collections API is a shadow of its former self, but the most-used collections—vectors, hashtables, and stacks—still remain.

Changes to Time Utilities

The CLDC time APIs are considerably streamlined as well. The notion of date formats is hidden entirely, having been wrapped in UI classes provided by the MIDP; the only facilities provided let you manage instances in time (`Date`) and time zones (`TimeZone`). `Calendar` provides the ability to get the current calendar, a millisecond timer, and other utilities.

Changes to Timer Behavior

Although not part of the CLDC proper, it's worth calling out the `Timer` and `TimerTask` classes here, as they're the only MIDP classes in the `java.util` hierarchy. Drawn from Java SE, the `Timer` class lets you schedule a `TimerTask` to be performed at some time in the future. As with Java SE, simply subclass `TimerTask` overriding `run`, and then use a `Timer` instance to schedule the operation. Of course, your MIDlet must be running; for background wake-up behavior, consider using an alarm (see the "Managing Startup Events and Alarms" section in Chapter 4).

Changes to the java.io Package

Table 2-4 lists the supported classes in the `java.io` package.

Table 2-4. *java.io Classes Supported by CLDC Version*

CLDC 1.0	CLDC 1.1
ByteArrayInputStream	ByteArrayInputStream
ByteArrayOutputStream	ByteArrayOutputStream
DataInput	DataInput (interface)
DataInputStream	DataInputStream
DataOutput	DataOutput (interface)
DataOutputStream	DataOutputStream
EOFException	EOFException
InputStream	InputStream
InputStreamReader	InputStreamReader
InterruptedIOException	InterruptedIOException
IOException	IOException
OutputStream	OutputStream
OutputStreamWriter	OutputStreamWriter
PrintStream	PrintStream
Reader	Reader
UnsupportedEncodingException	UnsupportedEncodingException
UTFDataFormatException	UTFDataFormatException
Writer	Writer

Heavily streamlined, this package omits access to the native file system, but it offers limited internationalization through the `InputStreamReader` and `OutputStreamWriter` classes, which accept an optional string indicating the character-encoding method.

Introducing Classes in the CLDC

The key introduction of the CLDC—which found its way into the CDC as well—is the GCF, which was described first in the CLDC 1.0 documentation and later provided with a migration path to Java SE in JSR 197.

The GCF provides a unified means to interact with networks over a variety of protocols, including but not limited to HTTP, TCP, and UDP. Using a URL schema for connection definition, clients obtain instances of a `Connection` subclass from the `Connector` factory and perform I/O over the `Connection` subclass. Table 2-5 lists the classes available. Chapter 12 describes the GCF in more detail, including examples of its use. In addition to these classes, packages such as the MIDP, the Foundation Profile for CDC, and the GCF for Java SE include additional connections, such as a connection that implements support for HTTP.

Table 2-5. *The javax.microedition.io Classes Supported by CLDC Version*

CLDC 1.0	CLDC 1.1
`Connection`	`Connection`
`ConnectionNotFoundException`	`ConnectionNotFoundException`
`Connector`	`Connector`
`ContentConnection`	`ContentConnection`
`Datagram`	`Datagram`
`StreamConnectionNotifier`	`StreamConnectionNotifier`
`InputConnection`	`InputConnection`
`OutputConnection`	`OutputConnection`
`DatagramConnection`	`DatagramConnection`
`StreamConnection`	`StreamConnection`

Besides its importance in providing CLDC applications with a path to communicating with the outside world, the GCF provides an object lesson in where Java ME is headed: into Java SE. As portable devices continue to become more full-featured, the innovations necessary to bring Java to those devices will likely become mainstream parts of the Java SE interface.

Changing the Java Class Library to Fit the CDC

The joy of working with the CDC is that there *are* no changes to fit the Java SE class hierarchy—what you see in Java SE is what you get in the Java ME CDC. As previously noted, however, the CDC is a superset of the CLDC; the GCF is included in the CDC as well. Table 2-6 shows the classes provided by the CDC in the `javax.microedition.io` package.

Table 2-6. *javax.microedition.io Library Additions to the CDC*

`Connection`
`ConnectionNotFoundException`
`Connector`
`ContentConnection`
`Datagram`
`HttpConnection`
`HttpsConnection*`
`SecureConnection*`
`StreamConnectionNotifier`
`InputConnection`
`OutputConnection`
`DatagramConnection`
`StreamConnection`

**Only if the device supports HTTPS.*

Note that the CDC supports HTTP out of the box, as opposed to the CLDC, which does not include HTTP except as provided by the MIDP.

Wrapping Up

If you take away one thing from this chapter, it's that the key difference between the CLDC and the CDC is what's included in terms of classes. The CLDC contains *far* fewer classes than the CDC, which is at parity with Java SE 1.4.2. Notable classes missing in the CLDC include most of the `java.util` hierarchy. Specifically, all of the collections except vectors, hashtables, and stacks are missing. Also, the CLDC provides streamlined time-management APIs, including a subset of `java.util.Date`, `java.util.Calendar`, and `java.util.TimeZone`. Finally, it possesses a simplified `java.io` hierarchy.

The CDC merges Java SE with the CLDC, meaning that both the CDC and the CLDC include the GCF, a media-independent hierarchy of interfaces that provide a factory of connections such as sockets, files, or other entities. The CLDC defers the definition of which protocols are supported by the GCF to the MIDP, while the CDC requires support for HTTP and HTTPS by default.

Neither the CLDC nor the CDC provides support for GUI development. For that, you need a profile such as the MIDP atop the configuration, which is the subject of Parts 2 and 3 of this book.

CHAPTER 3

Getting Started with the NetBeans IDE

Numerous tools are available for building Java ME applications, but the NetBeans integrated development environment (IDE) stands out as providing the best-of-breed support for the platform while remaining open for changes and extensions. If you're just starting out, the NetBeans software development kit (SDK) is the place to begin; if you're exploring the Java ME platform with thoughts of doing your work in another environment, many of the concepts you'll learn in this chapter still apply.

In this chapter, I begin with an introduction to the NetBeans IDE and explain how to install it for both Microsoft Windows and Linux. Next, I present a whirlwind tour of the NetBeans IDE; this is *very* brief, as the best way to learn the IDE is by using it. As a result, I spend considerable time using the IDE in two step-by-step tutorials: one to build your first CLDC/Java ME application, and the other to build your first CDC/AGUI application. After reading this chapter, you will understand the basics of laying out, editing, compiling, and packaging CLDC/Java ME and CDC applications using NetBeans. You'll also gain an understanding of the technology that you can bring to other environments, such as EclipseME, should you choose a different tool chain.

Selecting the NetBeans IDE

While a bevy of tools is available for doing Java ME work, it's a good idea to begin learning Java ME with the NetBeans IDE, for several reasons:

- It's free.
- It includes a GUI builder that lets you lay out complex screens quickly and with little effort, autogenerating the code behind the scenes.
- It includes full support for source-level debugging of your application.

- It supports round-trip development from source-code editing through deployment to a server for on-device testing.

- With the Mobility Pack (also free), it includes full support for Java and Java ME, including a handset emulator on which you can test your application before deploying to a device.

Of course, other IDEs for Java ME are available, including the also-free EclipseME. However, the NetBeans IDE with the Mobility Pack provides all you need to get started, but piecing together EclipseME requires that you download Eclipse and a specific version of the Sun Java Wireless Toolkit (also a free download) before downloading, installing, and configuring EclipseME. Moreover, EclipseME's resulting environment isn't quite as full-featured or well integrated as the NetBeans SDK, so I recommend starting with the NetBeans SDK instead.

Assuming you have Java SE Java Development Kit (JDK) version 4, 5, or 6 installed, installing the NetBeans IDE couldn't be easier—just head on over to the NetBeans web site at `http://www.netbeans.org/` and click the Download NetBeans IDE link. If you don't have the required Java SE JDK installed, you have two choices—surf over to Sun at `http://java.sun.com/` and download one, or download the NetBeans IDE bundled with the required JDK.

Once you've downloaded and installed the NetBeans IDE (the download provides a double-clickable installer), go back to `http://www.netbeans.org/` and find the Mobility Packs for both the CLDC/MIDP and the CDC. Download either, or both, depending on whether you want to target the CLDC/MIDP or the CDC, and run the installers provided.

However, there's one catch for Mac OS X developers—while NetBeans will run happily on your operating system of choice, the Mobility Pack will not. Fortunately, there's nothing to keep you from running the NetBeans IDE with the Mobility Pack under a virtual machine or Boot Camp with another operating system. (In fact, I created all of the examples in this book that way.) Simply consult the virtual machine provider's documentation to install Linux or Windows, and then proceed with these instructions inside the operating system of your choice.

■ **Note** I created all of the examples in this book using the NetBeans IDE version 5.5.1; later versions are available as this book goes to press. If you're using a later version of the NetBeans IDE than this, the screens and instructions that follow may be slightly different.

Finding Your Way Around the NetBeans IDE

The first time you launch the NetBeans IDE can be an overwhelming experience, especially if you're used to a different (or no) IDE. Figure 3-1 shows the NetBeans IDE while working on a typical project. Fortunately, the frustration quickly passes once you actually start using the environment, as the things you use frequently become natural, and the features you don't regularly use fade from your attention.

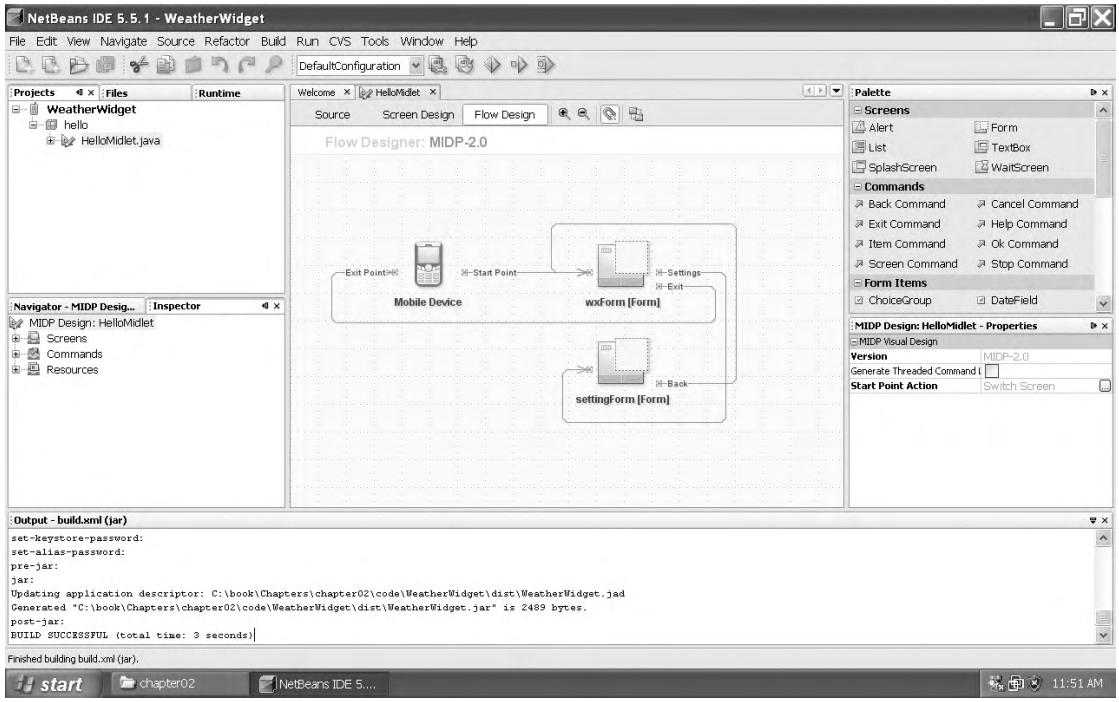


Figure 3-1. *The NetBeans IDE at work*

The NetBeans IDE is divided up into several windows. The following are some of the most important:

- *Projects/Files/Runtime*: Lets you inspect the package and file-system layout for your application
- *Navigator*: Lets you browse the members or class hierarchy of the class you're currently editing
- *Editor*: Lets you edit the design, layout, and source code of your application

- *Palette*: Visible when using the visual GUI designer (often referred to as Matisse in older documentation for the NetBeans IDE) with items you can drop onto the visual GUI editor window

- *Properties*: Lets you edit various properties of the item you have selected when working in the visual GUI designer

- *Output*: Shows the results of builds, the system log during execution, and so forth

Like other IDEs, one of the things you'll have to get used to is that these windows can come and go; for example, during application debugging, the region for the Output window is smaller, and the liberated space is used to contain another window with tabs showing watchpoints, local variables, and the call stack when the application is stopped at a breakpoint. Similarly, the Palette window may come and go depending on whether you're editing a visual layout (when it should be visible) or source code (when it may disappear). If you can't find a window you're looking for—a common experience, especially at first—just mouse on up to the Window menu bar and choose the window you're looking for.

The NetBeans IDE manages your source code as a *project*—that is, a collection of files within directories, including your source files and an input to Apache Ant for building your project. It's best not to fool with the directory hierarchy set out by the NetBeans SDK, but it helps to understand the purpose of each file and directory:

- `build.xml`: This file contains the build scripts for Ant that the NetBeans SDK uses to build your application.

- `nbproject`: This directory contains the files used by the IDE to manage your project itself.

- `src`: This directory contains the source code you write for your application.

- `test`: This optional directory contains the code for any unit tests you write.

- `resources`: This optional directory contains any binary resources for your application.

- `build`: This directory is used by the NetBeans IDE to contain your application's interbuild products, which consist of classes and other files.

- `dist`: This directory, generated by the NetBeans IDE, contains the generated Java executable files for your application.

If you're using revision-control software such as Concurrent Versions System (CVS) or Subversion (SVN)—and for any serious work, you should—you will want to have all of these under change control except the `build` and `dist` directories, which are generated at runtime by the NetBeans IDE when you perform builds. One slick feature of the IDE is

integrated support for both CVS and Subversion; CVS is built in, while SVN is available from the Update Center (choose Update Center from the Tools menu) as an optional module to install.

Creating Your First CLDC/MIDP Application

Assuming you've installed the CLDC Mobility Pack for the NetBeans IDE, you're ready to create your first CLDC application. The application you'll create in this section is the user interface for a simple weather display widget I call WeatherWidget; as you work through the book, you'll extend this application with features such as network support (to fetch real-time weather data) and persistence (to store the user's preferences). Figure 3-2 shows the application.

Figure 3-2. *The WeatherWidget application running in emulation*

The application consists of two screens: the main screen, which shows the current weather as obtained from a server, and a settings screen, which lets you enter a city and state in the United States. For this chapter, the weather data and location data are static; in Chapter 6, you will learn how to persist user settings, and in Chapters 12 and 13, you'll see various ways of interfacing with the Web (including using web services) to obtain real-time data for your application.

Walking Through the Creation of WeatherWidget

To create the WeatherWidget application, follow the steps in this section.

■ **Note** If you'd rather skim these steps before looking at the resulting project, you can find the source code that results from this sequence of steps in the `WeatherWidget` directory of the sample code for Chapter 3, found in the Source Code/Download area of the Apress web site (`http://www.apress.com`).

Creating the Project and Forms for the Screen

In this section, you'll learn how to create the project containing the application, as well as the forms that implement the screens of the application.

1. Within the NetBeans IDE, choose CREATE NEW PROJECT from the Welcome tab of the editor.

2. From the dialog that appears, choose Mobile from the Categories column, and choose Mobile Application from the Projects column. Click Next.

3. Enter **WeatherWidget** for the application name, and select a location for the project directory. Leave both Set as Main Project and Create Hello MIDlet ticked. Click Next.

4. Leave the defaults set, and click Finish to finish the New Project wizard.

5. With the wizard complete, the NetBeans IDE brings you to the Flow Designer (in the Editor window), which Figure 3-3 shows. Here you map out the flow between the screens of your application. Begin by selecting and renaming the helloForm instance you created to **wxForm** in the Flow Designer.

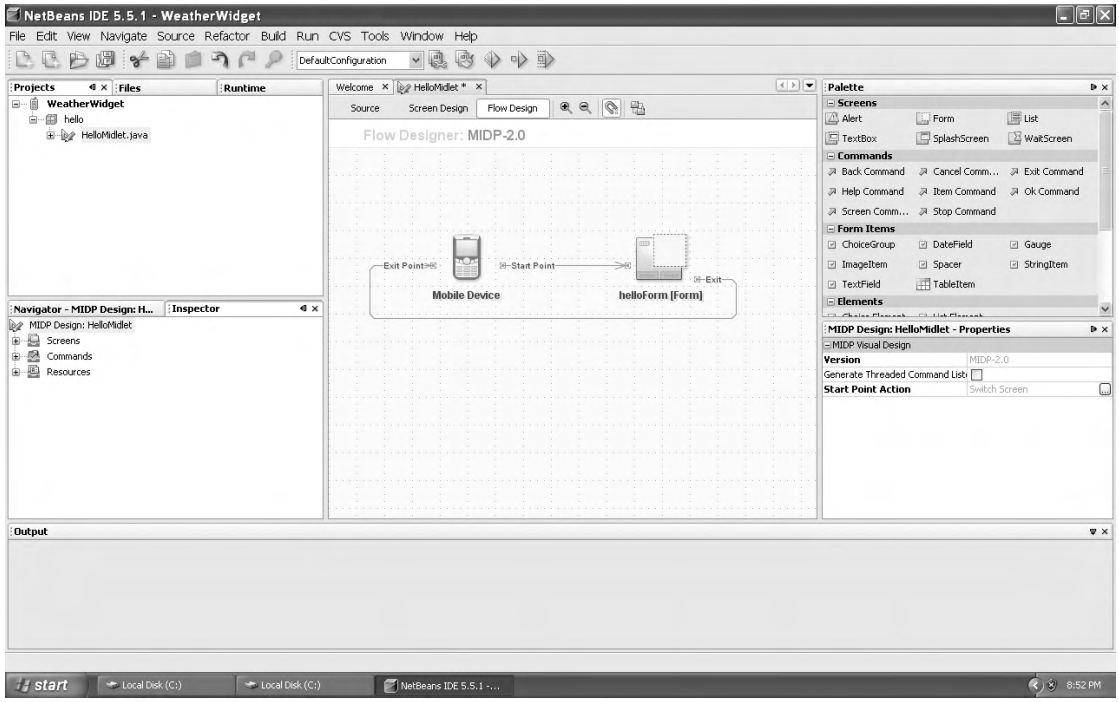


Figure 3-3. *The Flow Designer after renaming the first form*

6. Drag another Form screen to the Flow Designer; rename it to **settingForm**.

Wiring the Screen Transitions

In this section, you'll learn how to add screen transitions to the application. Follow these steps:

1. Now add and wire up the commands that transition between screens. From the Palette window on the right-hand side, drag an item named Ok Command to the wxForm screen in the Flow Designer.

2. From the Palette window, drag a Back Command item to the settingForm form.

3. Select the wxForm form and choose the Screen Designer by selecting Screen Design near the top of the Editor window. You will see the Screen Designer for the first form, similar to Figure 3-4.

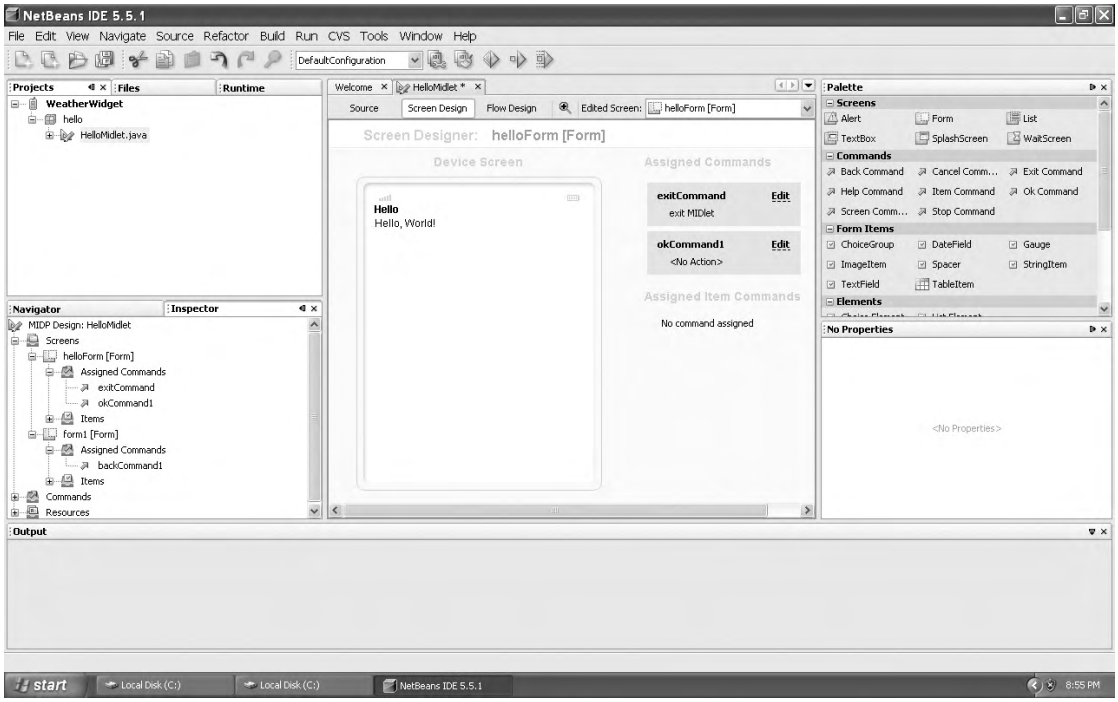


Figure 3-4. *The Screen Designer for the first form*

4. Click okCommand1 on the right side of the Editor window; edit the name to be **okCommand** in the Properties window. Also, in Properties, change its label to be **Settings**.

5. Click Edit in the okCommand box in the Editor window. In the window that appears, change its action to "Switch to screen" and set the settingForm form as its target (see Figure 3-5). Click OK.

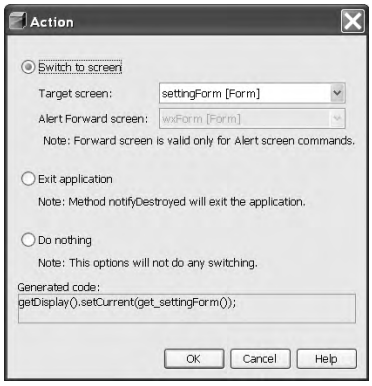


Figure 3-5. *The Action window, linking an item to a new screen*

6. Choose the settingForm form in the Edited Screen: combo box to transition to the other screen.

7. Rename the backCommand1 command to **backCommand** and set its action to "Switch to screen" with the wxForm form as the target.

8. Return to the Flow Designer. You should now have the flow as shown in Figure 3-6.

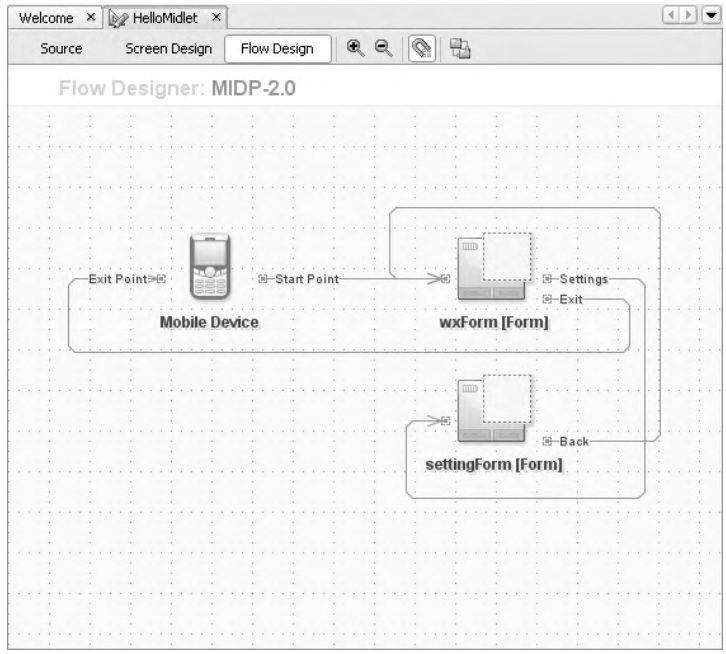


Figure 3-6. *The completed application flow in the Flow Designer*

9. While still in the Flow Designer, double-click the wxForm form icon to return to the Screen Designer.

Designing the Screens

In this section, you'll design the application's screens, adding components with which the user will interact. Follow these steps:

1. Click the helloStringItem item (the only widget on the screen in the Editor window). In the Properties window, make its label **Location** and its text **Berkeley, CA**, and change its name to **locationItem**.

2. Click Spacer from the Palette window's Form Items group to drag a spacer to the screen.

3. Adjust the spacer's height to ten pixels by clicking the button labeled "…" next to Minimum Size in the Editor window and changing its width in the pop-up window that appears.

4. Click StringItem from the Palette window's Form Items group to drag a string item to the screen.

5. Click the new StringItem item, and in the Properties pane, change its name to **wxItem**, its label to **Forecast**, and its contents to simply the word **Sunny**. The Screen Designer should now look like Figure 3-7.

Figure 3-7. *The completed widget screen in the Screen Designer*

6. Now add the items to the Settings screen. To begin, choose settingForm from the Edit Screen menu.

7. Click TextField to drag a text field to the screen. In the Properties pane, name it **cityField**, change its label to **City**, and change its contents to **Berkeley**.

8. Click Spacer to drag a spacer to the screen, and adjust its height to ten pixels.

9. Click TextField again to drag a second text field to the screen. In the Properties pane, name it **stateField**, change its label to **State**, and change its contents to **CA**.

10. Now set the name of the widget itself. Select the project, go to the top-level File menu, and choose WeatherWidget Properties.

11. Choose MIDlets from the right-hand pane, and rename the MIDlet to **Weather**.

Building and Running for the First Time

Build and run your application by clicking the green arrow. In a few seconds, you'll see the Sun Java Wireless Toolkit emulator. You can launch the application and explore the UI you created, or you can quit the application.

■ **Tip** If the emulator launches and then immediately exits, check your antivirus and firewall applications to ensure that it's not blocking the execution of the emulator.

Let's take a look at your handiwork. Listing 3-1 shows the code you created, largely through manipulating the Flow Designer and Screen Designer.

Listing 3-1. *Code Generated by Using the Flow Designer and Screen Designer*

```
/*
 * HelloMIDlet.java
 *
 * Created on November 19, 2007, 7:58 PM
 */

package com.apress.rischpater.weatherwidget;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 *
 * @author Ray Rischpater
 */
public class HelloMIDlet extends MIDlet implements CommandListener {
```

```
    /**
     * Creates a new instance of HelloMIDlet
     */
    public HelloMIDlet() {
    }

    private Form wxForm;//GEN-BEGIN:MVDFields
    private StringItem locationItem;
    private Command exitCommand;
    private Form settingForm;
    private Command okCommand;
    private Command backCommand;
    private Spacer spacer1;
    private StringItem wxItem;
    private StringItem stringItem1;
    private Spacer spacer2;
    private StringItem stringItem2;//GEN-END:MVDFields

//GEN-LINE:MVDMethods

    /** This method initializes UI of the application.//GEN-BEGIN:MVDInitBegin
     */
    private void initialize() {//GEN-END:MVDInitBegin
        // Insert pre-init code here
        getDisplay().setCurrent(get_wxForm());//GEN-LINE:MVDInitInit
        // Insert post-init code here
    }//GEN-LINE:MVDInitEnd

    /** Called by the system to indicate that a command has been invoked on➥
 a particular displayable.//GEN-BEGIN:MVDCABegin
     * @param command the Command that ws invoked
     * @param displayable the Displayable on which the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {➥
//GEN-END:MVDCABegin
        // Insert global pre-action code here
        if (displayable == wxForm) {//GEN-BEGIN:MVDCABody
            if (command == exitCommand) {//GEN-END:MVDCABody
                // Insert pre-action code here
                exitMIDlet();//GEN-LINE:MVDCAAction3
                // Insert post-action code here
            } else if (command == okCommand) {//GEN-LINE:MVDCACase3
```

```
                // Insert pre-action code here
                getDisplay().setCurrent(get_settingForm());➥
//GEN-LINE:MVDCAAction12
                // Insert post-action code here
            }//GEN-BEGIN:MVDCACase12
        } else if (displayable == settingForm) {
            if (command == backCommand) {//GEN-END:MVDCACase12
                // Insert pre-action code here
                getDisplay().setCurrent(get_wxForm());➥
//GEN-LINE:MVDCAAction14
                // Insert post-action code here
            }//GEN-BEGIN:MVDCACase14
        }//GEN-END:MVDCACase14
        // Insert global post-action code here
}//GEN-LINE:MVDCAEnd

    /**
     * This method should return an instance of the display.
     */
    public Display getDisplay() {//GEN-FIRST:MVDGetDisplay
        return Display.getDisplay(this);
    }//GEN-LAST:MVDGetDisplay

    /**
     * This method should exit the midlet.
     */
    public void exitMIDlet() {//GEN-FIRST:MVDExitMidlet
        getDisplay().setCurrent(null);
        destroyApp(true);
        notifyDestroyed();
    }//GEN-LAST:MVDExitMidlet

    /** This method returns instance for wxForm component and should be called➥
 instead of accessing wxForm field directly.//GEN-BEGIN:MVDGetBegin2
     * @return Instance for wxForm component
     */
    public Form get_wxForm() {
        if (wxForm == null) {//GEN-END:MVDGetBegin2
```

```
            // Insert pre-init code here
            wxForm = new Form(null, new Item[] {//GEN-BEGIN:MVDGetInit2
                get_locationItem(),
                get_spacer1(),
                get_wxItem()
            });
            wxForm.addCommand(get_exitCommand());
            wxForm.addCommand(get_okCommand());
            wxForm.setCommandListener(this);//GEN-END:MVDGetInit2
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd2
        return wxForm;
    }//GEN-END:MVDGetEnd2

    /** This method returns instance for locationItem component and should➥
 be called instead of accessing locationItem field directly.➥
//GEN-BEGIN:MVDGetBegin4
     * @return Instance for locationItem component
     */
    public StringItem get_locationItem() {
        if (locationItem == null) {//GEN-END:MVDGetBegin4
            // Insert pre-init code here
            locationItem = new StringItem("Location", "Berkeley, CA");➥
//GEN-LINE:MVDGetInit4
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd4
        return locationItem;
    }//GEN-END:MVDGetEnd4

    /** This method returns instance for exitCommand component and should ➥
be called instead of accessing exitCommand field directly.➥
//GEN-BEGIN:MVDGetBegin5
     * @return Instance for exitCommand component
     */
    public Command get_exitCommand() {
        if (exitCommand == null) {//GEN-END:MVDGetBegin5
            // Insert pre-init code here
            exitCommand = new Command("Exit", Command.EXIT, 1);➥
//GEN-LINE:MVDGetInit5
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd5
        return exitCommand;
    }//GEN-END:MVDGetEnd5
```

```
    /** This method returns instance for settingForm component and should ➥
be called instead of accessing settingForm field directly.➥
//GEN-BEGIN:MVDGetBegin10
     * @return Instance for settingForm component
     */
    public Form get_settingForm() {
        if (settingForm == null) {//GEN-END:MVDGetBegin10
            // Insert pre-init code here
            settingForm = new Form(null, new Item[] {//GEN-BEGIN:MVDGetInit10
                get_stringItem1(),
                get_spacer2(),
                get_stringItem2()
            });
            settingForm.addCommand(get_backCommand());
            settingForm.setCommandListener(this);//GEN-END:MVDGetInit10
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd10
        return settingForm;
    }//GEN-END:MVDGetEnd10

    /** This method returns instance for okCommand component and should be➥
 called instead of accessing okCommand field directly.➥
//GEN-BEGIN:MVDGetBegin11
     * @return Instance for okCommand component
     */
    public Command get_okCommand() {
        if (okCommand == null) {//GEN-END:MVDGetBegin11
            // Insert pre-init code here
            okCommand = new Command("Settings", Command.OK, 1);➥
//GEN-LINE:MVDGetInit11
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd11
        return okCommand;
    }//GEN-END:MVDGetEnd11

    /** This method returns instance for backCommand component and should ➥
be called instead of accessing backCommand field directly.➥
//GEN-BEGIN:MVDGetBegin13
     * @return Instance for backCommand component
     */
    public Command get_backCommand() {
        if (backCommand == null) {//GEN-END:MVDGetBegin13
```

```
            // Insert pre-init code here
            backCommand = new Command("Back", Command.BACK, 1);➥
//GEN-LINE:MVDGetInit13
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd13
        return backCommand;
    }//GEN-END:MVDGetEnd13

    /** This method returns instance for spacer1 component and should be ➥
called instead of accessing spacer1 field directly.➥
//GEN-BEGIN:MVDGetBegin15
     * @return Instance for spacer1 component
     */
    public Spacer get_spacer1() {
        if (spacer1 == null) {//GEN-END:MVDGetBegin15
            // Insert pre-init code here
            spacer1 = new Spacer(1000, 10);//GEN-LINE:MVDGetInit15
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd15
        return spacer1;
    }//GEN-END:MVDGetEnd15

    /** This method returns instance for wxItem component and should be called ➥
instead of accessing wxItem field directly.//GEN-BEGIN:MVDGetBegin16
     * @return Instance for wxItem component
     */
    public StringItem get_wxItem() {
        if (wxItem == null) {//GEN-END:MVDGetBegin16
            // Insert pre-init code here
            wxItem = new StringItem("Forecast", "Sunny.");//GEN-LINE:MVDGetInit16
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd16
        return wxItem;
    }//GEN-END:MVDGetEnd16

    /** This method returns instance for stringItem1 component and should ➥
be called instead of accessing stringItem1 field directly.➥
//GEN-BEGIN:MVDGetBegin23
     * @return Instance for stringItem1 component
     */
    public StringItem get_stringItem1() {
        if (stringItem1 == null) {//GEN-END:MVDGetBegin23
```

```
            // Insert pre-init code here
            stringItem1 = new StringItem("Location", "Berkeley, CA");➥
//GEN-LINE:MVDGetInit23
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd23
        return stringItem1;
    }//GEN-END:MVDGetEnd23

    /** This method returns instance for spacer2 component and should be called ➥
instead of accessing spacer2 field directly.//GEN-BEGIN:MVDGetBegin24
     * @return Instance for spacer2 component
     */
    public Spacer get_spacer2() {
        if (spacer2 == null) {//GEN-END:MVDGetBegin24
            // Insert pre-init code here
            spacer2 = new Spacer(1000, 10);//GEN-LINE:MVDGetInit24
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd24
        return spacer2;
    }//GEN-END:MVDGetEnd24

    /** This method returns instance for stringItem2 component and should ➥
be called instead of accessing stringItem2 field directly.➥
//GEN-BEGIN:MVDGetBegin25
     * @return Instance for stringItem2 component
     */
    public StringItem get_stringItem2() {
        if (stringItem2 == null) {//GEN-END:MVDGetBegin25
            // Insert pre-init code here
            stringItem2 = new StringItem("Forecast\n", "Sunny");➥
//GEN-LINE:MVDGetInit25
            // Insert post-init code here
        }//GEN-BEGIN:MVDGetEnd25
        return stringItem2;
    }//GEN-END:MVDGetEnd25

    public void startApp() {
        initialize();
    }

    public void pauseApp() {
    }
```

```
    public void destroyApp(boolean unconditional) {
    }

}
```

Wow! That's a lot of code for so little typing. As you browse the NetBeans IDE, you'll notice two things. First, large swaths of the code are on a blue background; the IDE autogenerates and maintains these blocks of code. Second, if you compare the appearance of these lines with the code shown in Listing 3-1, you'll notice that the listing has many lines with comments like `//GEN-BEGIN`, `//GEN_LINE`, and `//GEN_END`, followed by unique identifiers. The NetBeans IDE inserts these comments and uses them to tag the code it creates and maintains; if you edit the source code, be careful not to edit these lines, as you will lose your changes when you go back to use the Screen Designer and Flow Designer again.

While I go into more detail about the structure of MIDlets in Chapter 4, it's worth your time for me to point out now a few things about the code the NetBeans IDE generated for us. Just like traditional applets and applications, MIDlets have a specific life cycle, although it's a little different than for applets:

- *Construction*: When the application management system launches a MIDlet, an instance is created, resulting in the runtime invoking the MIDlet's constructor. The MIDlet is now said to be *paused*.

- *Active*: When the application manager calls the MIDlet's `startApp` method, the MIDlet is *active* and running normally.

- *Paused*: At any time, the application manager can pause the MIDlet to permit another application access to the handset (such as for an incoming call) by invoking `pauseApp`. Alternatively, a MIDlet can place itself in the same state by invoking `notifyPaused`. From this state, the application manager can resume the application by invoking `startApp` again.

- *Destroyed*: The application manager can terminate the MIDlet's execution at any time by calling `destroyApp`. Alternatively, the MIDlet can destroy itself by calling `notifyDestroyed`.

Figure 3-8 shows the MIDlet life cycle.

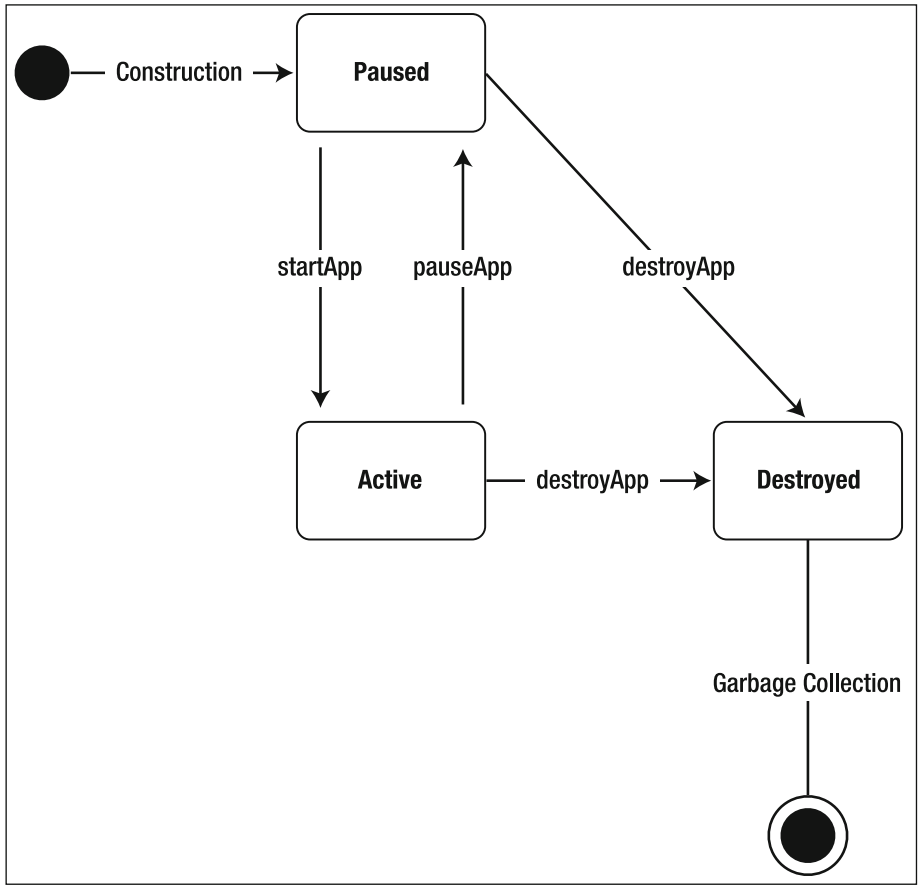


Figure 3-8. *The MIDlet life cycle*

As is generally the case, WeatherWidget doesn't do any object creation at construction time; instead, object creation is deferred until the application starts. The application manager–invoked `startApp` delegates the creation of items for the GUI to initialize and sets the display's current form to the `wxForm` field. The application does nothing on application pause or destruction, instead relying on the Java garbage collector upon destruction to reclaim the memory used by the application forms.

The MIDP environment provides a generalized notion of events; instead of handling events such as specific key presses, the runtime provides abstractions such as *OK*, *back*, and *help*. This abstraction permits the MIDP to run on a wide variety of devices, from touchscreen-only devices, to traditional phones with a four-way navigation pad and two soft keys, to speech-driven user interfaces. The system sends these commands to the MIDlet using the `commandAction` method, which simply switches on the incoming command to determine what screen to show or whether to exit completely (on an `exitCommand`).

Building CLDC/MIDP Applications

The NetBeans IDE uses the same Java compiler to build CLDC applications and other applications; behind the scenes, it uses a special option (`-bootpathoption`) to redirect the compiler to use different fundamental classes for the CLDC. This is important, because as you remember from the first chapter (and learned in detail in the previous chapter), not all classes and methods in the base hierarchy are provided as part of the CLDC.

Build options—especially optimization and obfuscation—play a big part in building for Java ME devices, for two reasons. First, these are commercial applications, distributed to a wide audience, so of course you want to keep your intellectual property away from the prying eyes of a decompiler. Second, and more important, optimization and obfuscation result in a smaller application, meaning that it takes less time to transfer to the target hardware and uses less memory on the target hardware. This is because obfuscation renames classes, member variables, and method names to shorter names, removing unused classes, methods, and member variables in the process. The NetBeans IDE includes the popular open source ProGuard obfuscator, which you can control using the Obfuscating panel in the Project Properties window. When compiling, you might want to test both optimized and nonoptimized builds for memory and time performance, too; go to the "Compile with Optimization" item in the Compiling panel of your project's properties. Finally, get in the habit of shipping release builds with no debugging information; doing so will lead to smaller binaries.

To manage all of this, the NetBeans SDK provides the notion of a *project configuration*, which is a collection of project options that includes the target platform, the application description and packaging (see the next section), and the build options. You create new project configurations in the Project Properties window by clickin the Manage Configurations button in the upper-right corner. Doing this brings you to a list of configurations from which you can add and remove configurations; in turn, project properties are keyed by the configuration you select in the Project Configuration window, shown in Figure 3-9. As you can tell from the figure, I like to have three project configurations: one for debugging, one for release, and one for a default configuration. More complex projects may require additional configurations to manage specific builds for particular hardware targets or other variables.

If you're familiar with the Java build process, you may have seen an additional step during the build you performed in the previous section. Take a close look at the output log, and you'll see lines after the obfuscation step labeled `pre-preverify`, `preverify`, and `post-preverify`.

As you recall from the first chapter, unlike the CDC or the standard JVM, the virtual machine used by the CLDC delegates some of the more expensive bytecode verification to the build process. This occurs *after* code obfuscation, and it's the last step prior to packaging your application. The preverify tool inlines subroutines in each class file and adds information to each stack frame to make it easier for the runtime virtual machine to perform necessary type checking and other bytecode verification.

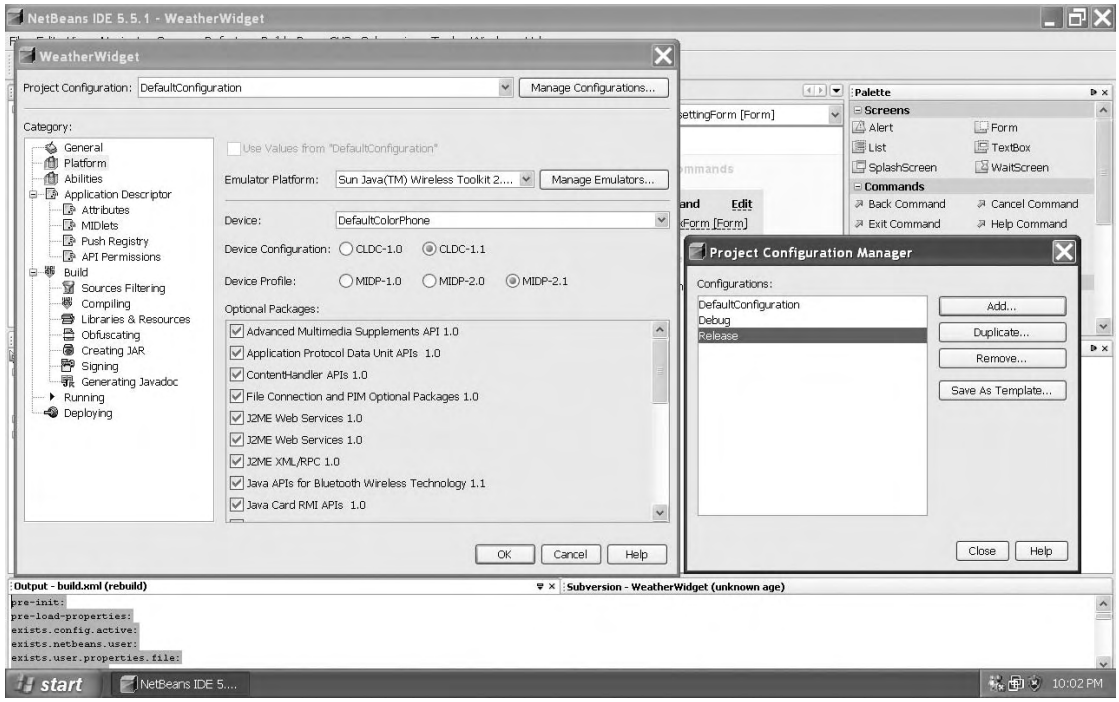


Figure 3-9. *The Project Properties dialog, titled WeatherWidget (left), and the Project Configuration Manager dialog (right)*

If you want to build your application using a tool chain other than the NetBeans IDE, you will need this preverify tool. Simply install a copy of the Sun Java Wireless Toolkit (available from `http://java.sun.com/`). Other tool chains, such as EclipseME, leverage the preverify tool from the Sun Java Wireless Toolkit or provide their own.

Packaging and Executing CLDC/MIDP Applications

On a device, an application manager provides services to MIDlets; for example, it downloads MIDlets, launches and terminates MIDlets, shares system resources with MIDlets, and so forth. A MIDlet presents itself to the application manager as two files: a JAD file that describes the application, and a Java Archive (JAR) file that contains the bytecodes for the application along with any required resources. In fact, more than one MIDlet can be packaged in a JAD/JAR pair; this is called a *suite*, and you must have entries in the JAD for each MIDlet in the suite.

A JAD file is a name-value pair of attributes, such as the one generated by the NetBeans IDE for the WeatherWidget application that you see in Listing 3-2.

Listing 3-2. *The WeatherWidget JAD File*

```
MIDlet-1: Weather,,com.apress.rischpater.weatherwidget.WeatherWidget
MIDlet-Jar-Size: 2858
MIDlet-Jar-URL: WeatherWidget.jar
MIDlet-Name: WeatherWidget
MIDlet-Vendor: Ray Rischpater
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.1
```

This is a bare-bones JAD file, built by the NetBeans IDE from the project properties I defined when creating the application. Relevant fields include

- `MIDlet-`*n*: The name, the path in the JAR file to the icon, and the class name for the *n*th MIDlet in the MIDlet suite.
- `MIDlet-Jar-Size`: The size in bytes of the MIDlet suite's JAR file. This *must* match the actual size of the JAR file, or many devices won't accept the JAR file.
- `MIDlet-Jar-URL`: The URL of the JAR file for the MIDlet.
- `MIDlet-Name`: The name of the MIDlet suite.
- `MIDlet-Vendor`: The name of the vendor of the MIDlet suite.
- `MIDlet-Version`: The version number of the MIDlet suite.
- `MicroEdition-Configuration`: The version of the Java ME VM (CDC or CDLC and version number) required by the MIDlet suite.
- `MicroEdition-Profile`: The profile used by the MIDlet, including its version number.

You set these attributes in the Project Properties window in the Applet Descriptor section. The Applet Descriptor section has separate subsections for generic attributes and MIDlet attributes. As part of the JAD file, you also specify items for the *push registry* and *API permissions*—two kinds of information new to MIDP 2.0 (JSR 118).

The push registry and the corresponding Push API let the application manager activate your MIDlet based on incoming events such as an inbound network connection, an SMS message, or a timer alarm. The information you provide for a push-registry entry is a tuple consisting of the inbound endpoint, the class name to receive the push, and a filter indicating valid originators of the push. I discuss this in more detail in Chapter 14. Push-registry entries are specified using the name `MIDlet-Push-`*n*, where *n* is an integer indicating a unique push entry.

API permissions indicate that a specific permission is required when using a restricted API, such as the socket connection interface. To do this, your MIDlet must have the permission `javax.microedition.io.Connector.socket`, which indicates permission to use this API. Note that this is a necessary but not sufficient requirement to actually use the API; the runtime may prompt the user for approval or deny the operation on the basis of the origin of the application. API permissions are strings named after the package and class of the restricted API. (As I introduce APIs in subsequent chapters, I note the permissions required for restricted APIs.) Two JAD fields indicate permissions: `MIDlet-Permissions` and `MIDlet-Permissions-Opt`. The first names are the required permissions, and the second names are the optional permissions that MIDlets in the suite may use.

■ **Caution** Just providing permission in your JAD file *doesn't* entitle you to access the restricted APIs that require that permission. As I note in Chapter 1 in the section titled "Marketing and Selling Your Application," your application may also require a signature from a third party, such as Java Verified or a wireless operator, to use the API you require.

A final aspect of packaging your application is *signing*—that is, the process of cryptographically signing your application to prove that you're the originator of the application. Signing is the final link in the chain of the Java ME permission mechanism; signatures indicate that applications come from sources with a particular level of trust, and depending on the signatures an application bears, the application may be granted additional permissions to operate. Begin with a certificate provided by a well-known provider such as VeriSign, and import this into the NetBeans keystore using the Security Manager. To do this, follow these steps:

1. Obtain a certificate from an authority trusted by the operators on which you'll be deploying your application.

2. Import the certificate into your keystore using the keytool application included with the JDK. Enter this command from a command line:

```
% keytool -import -alias your_alias_for_certificate
         -file your_certificate_file
          -keystore your_keystore.ks
```

3. In the NetBeans IDE, add your keystore to the IDE by going to Project Properties ➤ Signing ➤ Security Manager ➤ Add Keystore.

4. Still in the Signing pane, tick the Sign Distribution box and select the alias of the certificate you just imported.

To test access to APIs, you can often use a self-signed JAR file by exporting your certificate to your handset. However, how you import the certificate depends on the device and the network you're using, so you should consult with the documentation that accompanies the device with which you're working.

Some devices let you transfer the JAD and JAR files via Bluetooth or a cable, but by far the most common way to get your code to a handset is over the Web. To do this, you should have access to a web server on the same network as your test device (in other words, don't rely on a corporate web server behind a firewall and a wireless terminal on an operator's network outside your firewall). Using the NetBeans IDE, you can transfer your JAD and JAR files to the server using Secure Copy Protocol (SCP) or another mechanism by right-clicking your application and choosing Deploy Project. From there, you can navigate to the URL of your JAD file using your device's application manager or the web browser. Next, on the device, download the JAD file, which triggers the installation of your application. Figure 3-10 shows the NetBeans Deployment Settings panel.

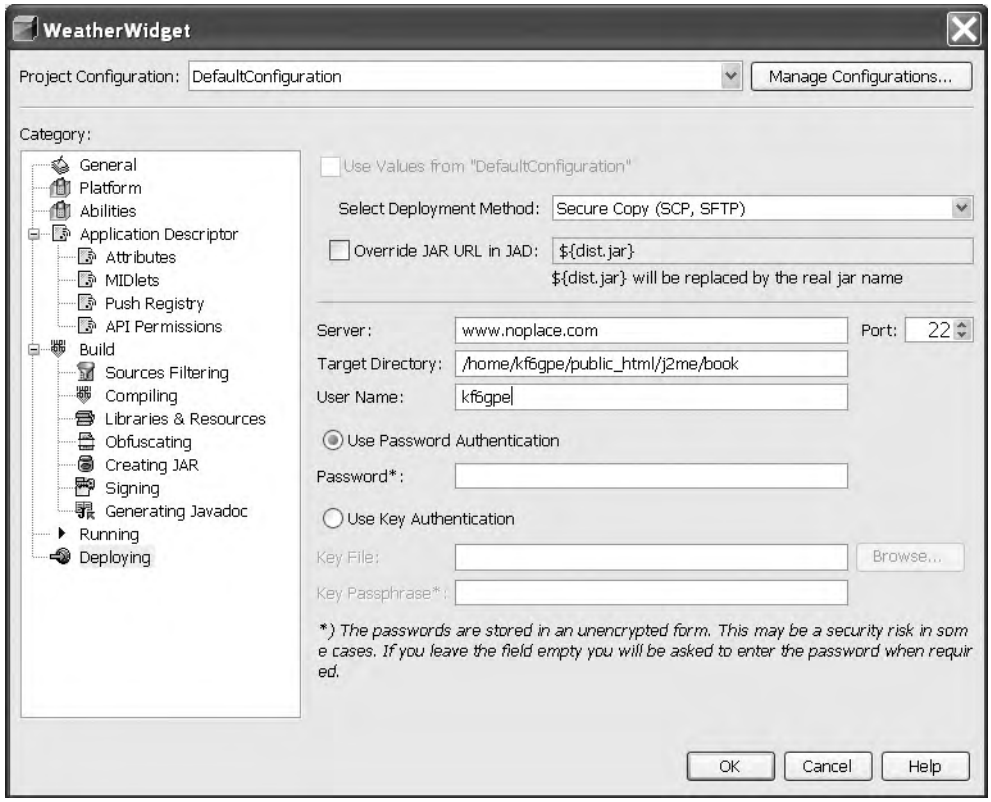


Figure 3-10. *Selecting how your application will be deployed by the NetBeans SDK*

Creating Your First CDC Application

As with your first CLDC application, writing your first CDC application requires that you install the NetBeans Mobility Pack for CDC before you begin working. WeatherApplet, the sample application you create here (shown in Figure 3-11), is the CDC analogue of the CDLC WeatherWidget application you created in the previous section. As you did with WeatherWidget, in this section you build only the UI, using the AGUI APIs described in JSR 209. A subset of Java Swing, you'll learn more about these APIs in Chapter 10.

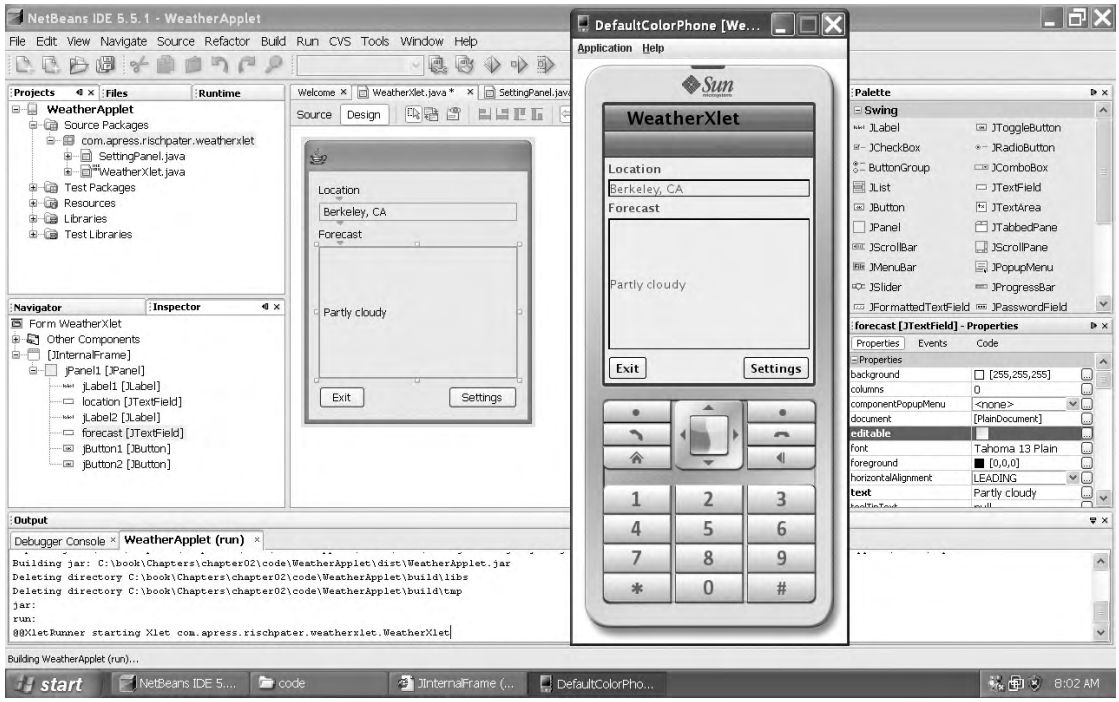


Figure 3-11. *The WeatherApplet application*

Walking Through the Creation of WeatherApplet

Perform the steps in the following sections to create WeatherApplet using the NetBeans IDE.

Creating the Project

Follow these steps to create the project for WeatherApplet:

1. Click CREATE NEW PROJECT in the Welcome pane.
2. Choose CDC from the Categories pane, and choose CDC Application from the Projects pane. Click Next.
3. Accept the default settings by clicking Next.
4. Enter the name **WeatherApplet** in the Project Name and Application Name fields. If Create Main Class is selected, uncheck it. Click Finish.

Creating the User Interface

Follow these steps to create the user interface:

1. In the Projects pane, right-click the Source Packages item and choose New ➤ AGUI Xlet Form, as shown in Figure 3-12.

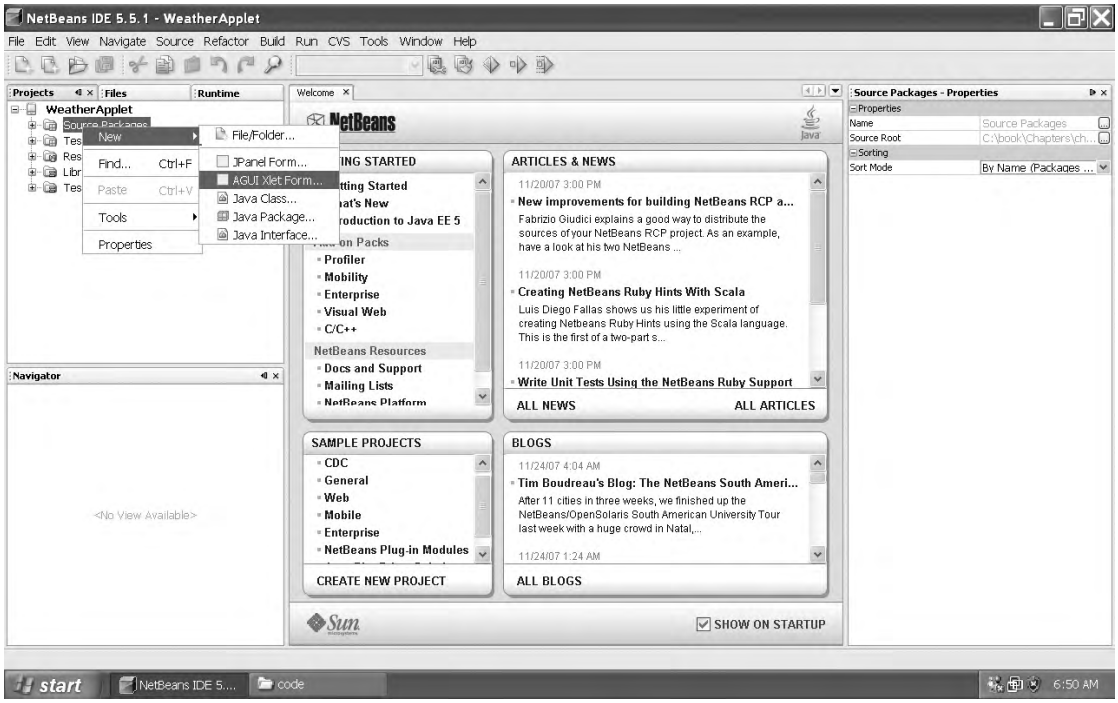


Figure 3-12. *Inserting a new Xlet form*

2. In the dialog that appears, name the form **WeatherXlet**. Place the form in a package if you like, too. Click Finish.

3. From the Palette pane on the right-hand side, click the `JPanel` item to drag out a JPanel control. Inside, place two `JLabels` and two `JTextFields`, along with two `JButton` controls, so that the form resembles Figure 3-13.

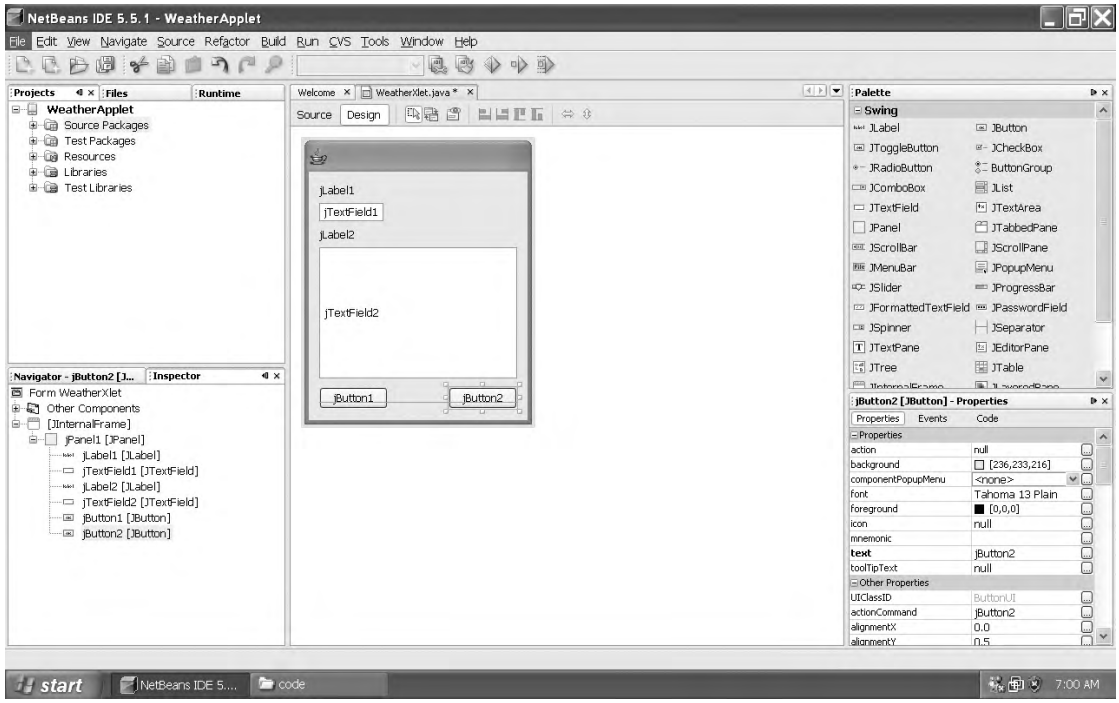


Figure 3-13. *Layout of the main form's components*

4. Change the fields and buttons and their contents to match Figure 3-14. You can do this by double-clicking and right-clicking items, or by using the Properties pane (by default, on the lower-right side of the display).

5. Change the name of `jTextField1` to **location** and the name of `jTextField2` to **forecast** using the Navigator panel on the lower-left side.

6. Make the two `JTextField` controls you just renamed neither focusable nor editable by unticking those attributes in the Properties pane for each item.

7. Now create the Settings panel. In the Projects pane, right-click the Source Packages item and choose New ➤ JPanel Form again.

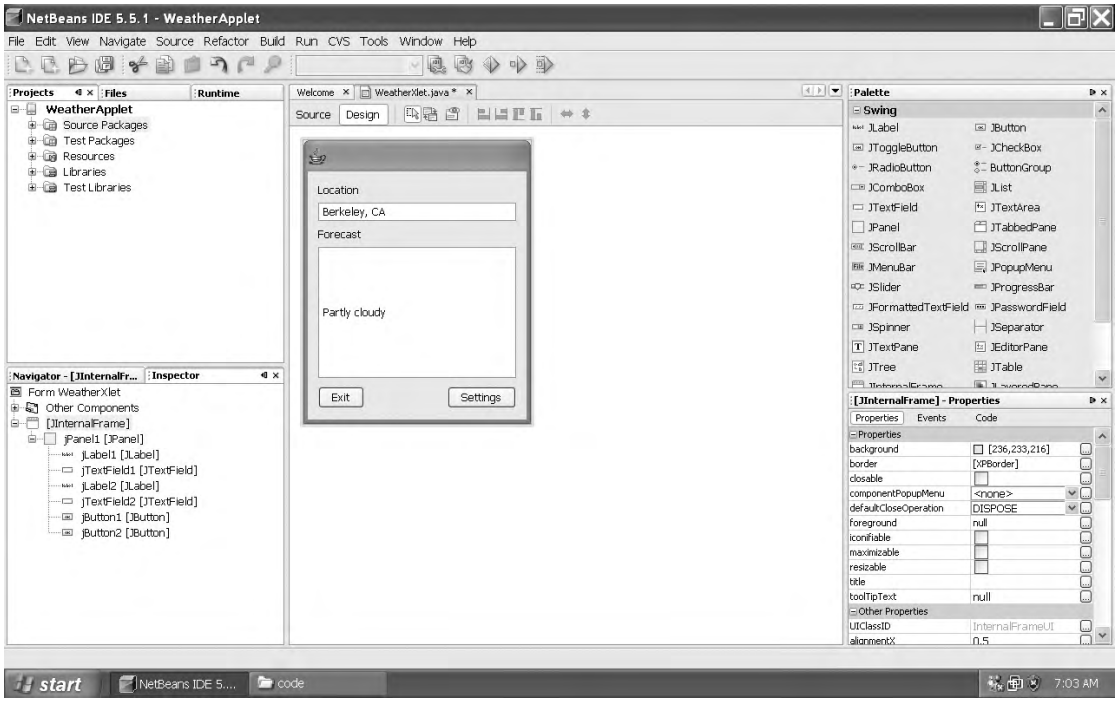


Figure 3-14. *Layout of the main form's components after you have renamed them*

8. In the dialog that appears, name the form **SettingPanel**. Place the form in the same package as the one you created (if any) in step 6.

9. Drag out and and name a `JPanel`, two `JLabels`, two `JTextFields`, and a `JButton` control to make the settings dialog shown in Figure 3-15.

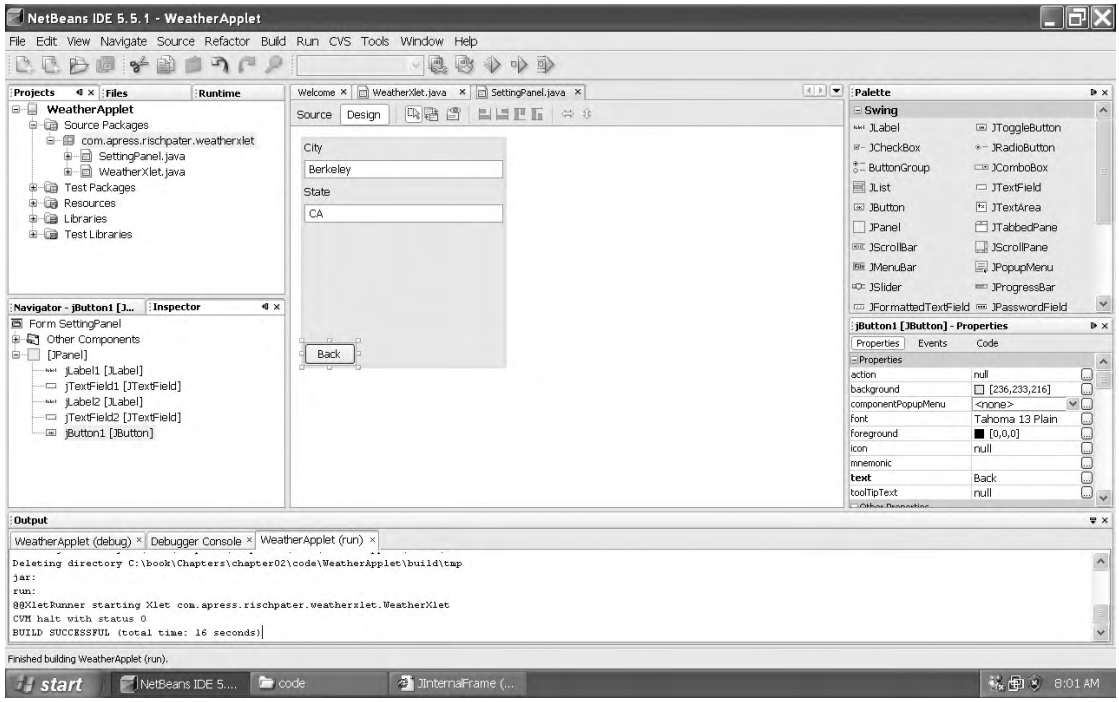


Figure 3-15. *Layout of the setting form's components*

Adding Actions to Wire the User Interface

Follow these steps to wire up the user interface:

1. Return to the WeatherXlet editor by selecting the WeatherXlet tab. Right-click the Settings button, and add an event by choosing Events ➤ Action ➤ actionPerformed. The code editor will open. Add the snippet in Listing 3-3 to the `actionPerformed` method.

Listing 3-3. *The actionPerformed Method for the Settings Button*

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    org.jdesktop.layout.GroupLayout layout =
      (org.jdesktop.layout.GroupLayout)getContentPane().getLayout();
    layout.replace( jPanel1, new SettingPanel());
}
```

2. Return to the Design view and add the `actionPerformed` event handler, shown in Listing 3-4, to the Exit button.

Listing 3-4. *The actionPerformed Method for the Exit Button*

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    exit();
}
```

3. Test the application by choosing Build Main Project from the Build menu. When the build is complete, you can run the application by choosing Run Main Project from the Run menu. When prompted for the main class, choose WeatherXlet.

Listing 3-5 shows the `WeatherXlet` class.

Listing 3-5. *The WeatherXlet Class*

```
/*
 * WeatherXlet.java
 *
 * Created on November 24, 2007, 6:57 AM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package com.apress.rischpater.weatherxlet;

import java.awt.Container;
import java.awt.EventQueue;
import javax.microedition.xlet.UnavailableContainerException;
import javax.microedition.xlet.XletContext;
import javax.microedition.xlet.XletStateChangeException;

/**
 *
 * @author Ray Rischpater
 */
public class WeatherXlet extends javax.swing.JInternalFrame implements➥
javax.microedition.xlet.Xlet {

    private XletContext context;            // our Xlet application context.
    private Container rootContainer;        // the root container of our screen.
```

```
    /** Creates new form WeatherXlet */
    public WeatherXlet() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    // <editor-fold defaultstate="collapsed" desc=" Generated Code ">➥
//GEN-BEGIN:initComponents
    private void initComponents() {
        jPanel1 = new javax.swing.JPanel();
        jLabel1 = new javax.swing.JLabel();
        location = new javax.swing.JTextField();
        jLabel2 = new javax.swing.JLabel();
        forecast = new javax.swing.JTextField();
        jButton1 = new javax.swing.JButton();
        jButton2 = new javax.swing.JButton();

        setFocusable(false);
        jLabel1.setText("Location");

        location.setEditable(false);
        location.setText("Berkeley, CA");
        location.setFocusable(false);
        location.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                locationActionPerformed(evt);
            }
        });

        jLabel2.setText("Forecast");

        forecast.setEditable(false);
        forecast.setText("Partly cloudy");
        forecast.setFocusable(false);
        forecast.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                forecastActionPerformed(evt);
            }
        });
```

```
        jButton1.setText("Exit");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        jButton2.setText("Settings");
        jButton2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton2ActionPerformed(evt);
            }
        });

        org.jdesktop.layout.GroupLayout jPanel1Layout = new org.jdesktop.➥
layout.GroupLayout(jPanel1);
        jPanel1.setLayout(jPanel1Layout);
        jPanel1Layout.setHorizontalGroup(
            jPanel1Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.➥
LEADING)
            .add(jPanel1Layout.createSequentialGroup()
                .add(jPanel1Layout.createParallelGroup(org.jdesktop.layout➥
.GroupLayout.LEADING)
                    .add(jLabel1)
                    .add(jLabel2))
                .addContainerGap(179, Short.MAX_VALUE))
            .add(forecast, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,➥
228, Short.MAX_VALUE)
            .add(jPanel1Layout.createSequentialGroup()
                .add(jButton1)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED,➥
96, Short.MAX_VALUE)
                .add(jButton2))
            .add(location, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,➥
228, Short.MAX_VALUE)
        );
        jPanel1Layout.setVerticalGroup(
            jPanel1Layout.createParallelGroup(org.jdesktop.layout.GroupLayout.➥
LEADING)
            .add(jPanel1Layout.createSequentialGroup()
                .add(jLabel1)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
```

```
                .add(location, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,➥
org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layou➥
t.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(jLabel2)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(forecast, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,➥
151, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED,➥
10, Short.MAX_VALUE)
                .add(jPanel1Layout.createParallelGroup(org.jdesktop.layout➥
.GroupLayout.BASELINE)
                    .add(jButton1)
                    .add(jButton2)))
        );

        org.jdesktop.layout.GroupLayout layout = new org.jdesktop.layout.➥
GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)

            .add(org.jdesktop.layout.GroupLayout.TRAILING, layout.create➥
SequentialGroup()
                .addContainerGap()
                .add(jPanel1, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,➥
org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addContainerGap())
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)

            .add(org.jdesktop.layout.GroupLayout.TRAILING, layout.createSe➥
quentialGroup()
                .addContainerGap()
                .add(jPanel1, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,➥
org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addContainerGap())
        );
        pack();
    }// </editor-fold>//GEN-END:initComponents
```

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_jButton1ActionPerformed
        exit();
    }//GEN-LAST:event_jButton1ActionPerformed

    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_jButton2ActionPerformed
        controller.showSettingPanel();
    }//GEN-LAST:event_jButton2ActionPerformed

    private void forecastActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_forecastActionPerformed
// TODO add your handling code here:
    }//GEN-LAST:event_forecastActionPerformed

    private void locationActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_locationActionPerformed
// TODO add your handling code here:
    }//GEN-LAST:event_locationActionPerformed


    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JTextField forecast;
    private javax.swing.JButton jButton1;
    private javax.swing.JButton jButton2;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JTextField location;
    // End of variables declaration//GEN-END:variables
    private WeatherController controller;
    private javax.swing.JPanel jPanel2;

    public void initXlet(final XletContext xletContext) throws XletStateChange➥
Exception {
        context = xletContext;
        if(rootContainer == null) {
            try {
                //This call to getContainer() tells the OS we want to be a➥
graphical app.
                rootContainer = context.getContainer();
            } catch (UnavailableContainerException e) {
                System.out.println("Ouch ! could not get our container!")
```

```
                // If we can't get the root container,
                // abort the initialization
                throw new XletStateChangeException( "Start aborted -- no➥
container: "
                        + e.getMessage() );
            }
        }
    }

    public void startXlet() throws XletStateChangeException {
        // Note: Swing thread constraints still apply in an Xlet... most➥
operations
        // need to be on the event thread, and this invokeLater does that.
        try {
            // using invokeAndWait to avoid writing synchronization code.
            // invokeLater would work just as well in most cases.
            EventQueue.invokeAndWait(new Runnable() {
                public void run() {
                    WeatherXlet.this.setVisible(true);
                    rootContainer.add(WeatherXlet.this);
                    // This is needed - or nothing will be displayed.
                    rootContainer.setVisible(true);
                }
            });
        } catch (Exception e) {
            System.out.println("Ouch - exception in invokeAndWait()");
            e.printStackTrace();
            exit();
        }
    }

    public void pauseXlet() {
        //This is pure overkill for this application, but is done to demonstate➥
the point.
        //We are freeing up our only resources (the screen), and we will rebuild➥
it when
        //we get started again. If you took out this block - the application➥
should still
        //run perfectly, and the screen should only be created once.
        try {
            // using invokeAndWait to avoid writing synchronization code.
            // invokeLater would work just as well in most cases.
```

```
            EventQueue.invokeAndWait(new Runnable() {
                public void run() {
                    rootContainer.remove(WeatherXlet.this);
                }
            });
        } catch (Exception e) {
            System.out.println("Ouch - exception in invokeAndWait()");
            e.printStackTrace();
            exit();
        }
    }

    public void destroyXlet(boolean b) throws XletStateChangeException {
        System.out.println("HelloXet.destroylet() - goodbye");
    }

    public void exit(){
        rootContainer.setVisible( false );
        context.notifyDestroyed();
    }
}
```

Listing 3-6 shows the `SettingPanel` class.

Listing 3-6. *The SettingPanel Class*

```
/*
 * SettingPanel.java
 *
 * Created on November 24, 2007, 7:55 AM
 */

package com.apress.rischpater.weatherxlet;

/**
 *
 * @author  Ray Rischpater
 */
public class SettingPanel extends javax.swing.JPanel {

    /** Creates new form SettingPanel */
    public SettingPanel() {
        initComponents();
    }
```

```
    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    // <editor-fold defaultstate="collapsed" desc=" Generated Code ">➥
//GEN-BEGIN:initComponents
    private void initComponents() {
        jLabel1 = new javax.swing.JLabel();
        jTextField1 = new javax.swing.JTextField();
        jLabel2 = new javax.swing.JLabel();
        jTextField2 = new javax.swing.JTextField();
        jButton1 = new javax.swing.JButton();

        jLabel1.setText("City");

        jTextField1.setText("Berkeley");

        jLabel2.setText("State");

        jTextField2.setText("CA");

        jButton1.setText("Back");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        org.jdesktop.layout.GroupLayout layout = new org.jdesktop.layout.➥
GroupLayout(this);
        this.setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
            .add(layout.createSequentialGroup()
                .add(layout.createParallelGroup(org.jdesktop.layout.Group➥
Layout.LEADING)
                    .add(jLabel1)
                    .add(jLabel2))
                .addContainerGap(199, Short.MAX_VALUE))
            .add(jTextField1, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,➥
229, Short.MAX_VALUE)
```

```
            .add(jTextField2, org.jdesktop.layout.GroupLayout.DEFAULT_SIZE,➥
229, Short.MAX_VALUE)
            .add(layout.createSequentialGroup()
                .add(jButton1)
                .addContainerGap())
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
            .add(layout.createSequentialGroup()
                .add(jLabel1)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(jTextField1, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,➥
org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layout.➥
GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(jLabel2)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(jTextField2, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,➥
org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layout.➥
GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED,➥
139, Short.MAX_VALUE)
                .add(jButton1))
        );
    }// </editor-fold>//GEN-END:initComponents

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_jButton1ActionPerformed
        controller.showMainPanel();
    }//GEN-LAST:event_jButton1ActionPerformed


    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JTextField jTextField1;
    private javax.swing.JTextField jTextField2;
    // End of variables declaration//GEN-END:variables
    WeatherController controller;
}
```

For each of these classes, the bulk of the implementation is in the `initComponents` method, which is responsible for creating the various user-interface controls and positioning them in the layout. Mercifully, all of this code is generated automatically, so you don't need to concern yourself with it in detail here.

The WeatherApplet application itself is an Xlet, similar to a MIDlet. Like MIDlets, Xlets have a well-defined life cycle in which the application can pass through five states, as shown in Figure 3-16. The NetBeans-generated code includes reference implementations for the Xlet method that implement the state transitions, saving you the need to write anything as you first get things going. You'll learn all about Xlets in Chapter 9.

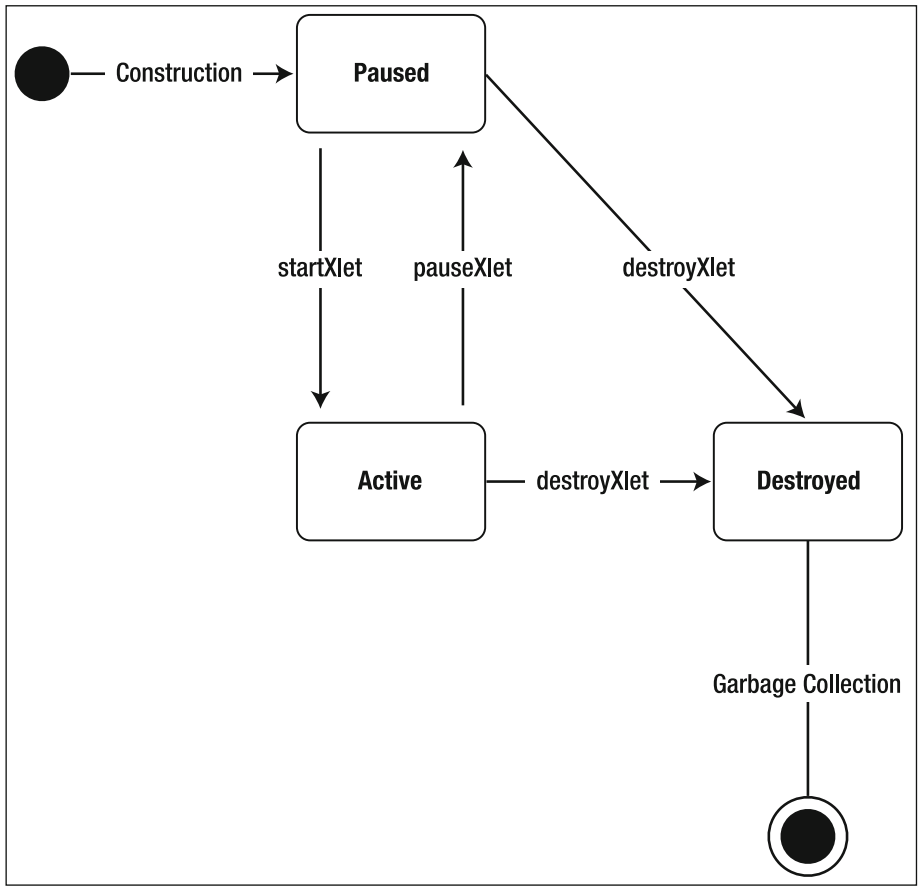


Figure 3-16. *The life cycle of an Xlet*

Wiring up the Back button in the `SettingPanel` class is trickier; to do this, create a controller that's responsible for managing transitions between the different panels of the user interface. The controller is responsible for hiding and showing each form in response to button clicks; both the Settings button and the Back button delegate screen

transitions to the controller. To create the controller class and hook it to the buttons, follow these steps:

1. Add the Java class `WeatherController` to the package (right-click the Source Packages item and choose New ➤ Java Class). Change its contents to read as shown in Listing 3-7. While you're there, you should change the package name, too.

Listing 3-7. *The WeatherController Class*

```
/*
 * WeatherController.java
 *
 * Created on November 24, 2007, 8:03 AM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package com.apress.rischpater.weatherxlet;
import javax.swing.JPanel;
import org.jdesktop.layout.GroupLayout;

/**
 *
 * @author Ray Rischpater
 */
public class WeatherController {
    private JPanel mainPanel, settingPanel;
    WeatherXlet xlet;

    GroupLayout layout;

    /** Creates a new instance of WeatherController */
    public WeatherController( WeatherXlet x) {
        xlet = x;
        layout = (GroupLayout)xlet.getContentPane().getLayout();
    }

    public void setMainPanel( JPanel m ) {
        mainPanel = m;
    }
```

```
    public void setSettingPanel( JPanel s ) {
        settingPanel = s;
    }

    public void showMainPanel() {
        layout.replace( settingPanel, mainPanel );
        xlet.pack();
    }

    public void showSettingPanel() {
        layout.replace( mainPanel, settingPanel );
        xlet.pack();
    }
}
```

2. Add the instance variables shown in Listing 3-8 to `WeatherXlet`.

Listing 3-8. *Declarations in WeatherXlet to Support the Controller*

```
private WeatherController controller;
private javax.swing.JPanel jPanel2;
```

3. Change `WeatherXlet`'s constructor to read as shown in Listing 3-9.

Listing 3-9. *Revising WeatherXlet's Constructor to Support the Controller*

```
public WeatherXlet() {
    initComponents();
    controller = new WeatherController( this );
    jPanel2 = new SettingPanel( controller );

    controller.setMainPanel( jPanel1 );
    controller.setSettingPanel( jPanel2 );
}
```

■ **Tip** Don't actually compile this code yet; you need to make the changes to the constructor in step 6 before your code is complete. You'll get an error if you attempt to build this code now.

4. Change the `actionPerformed` method for the Settings button to read as shown in Listing 3-10.

Listing 3-10. *The actionPerformed Method for the Settings Button*

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_jButton2ActionPerformed
    controller.showSettingPanel();
}
```

5. Add the following instance variable to `SettingPanel`:

```
WeatherController controller;
```

6. Change the `SettingPanel`'s constructor to read as shown in Listing 3-11.

Listing 3-11. *Revising the SettingPanel's Constructor*

```
public SettingPanel( WeatherController c ) {
    controller = c;
    initComponents();
}
```

7. Change the `SettingPanel`'s Back button's `actionPerformed` method to read as shown in Listing 3-12.

Listing 3-12. *Revising the SettingPanel's Back Button's actionPerformed Method*

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {➥
//GEN-FIRST:event_jButton1ActionPerformed
    controller.showMainPanel();
}
```

To see this working, you can simply build and run the application again, but this time, try using the source-level debugger by placing a breakpoint on the line that reads

```
layout.replace( mainPanel, settingPanel );
```

in the `showSettingPanel` of the `WeatherController` class. To do this, place the cursor on the line and click in the window margin on the left, or select Run ➤ Toggle Breakpoint from

the menu bar (or press Ctrl+F8). The line of code with the breakpoint will be highlighted in red, and a red square will appear in the window's left margin, indicating that execution will pause at that line during debugging. You can begin execution by choosing Run ➤ Debug Main Project (or pressing F5, or clicking the second of the three arrows in the toolbar). When you do this, execution will begin normally, but when you click the Settings button in the emulator, execution will stop at the breakpoint, letting you single-step through the `showSettingPanel`, examine variables and the call stack using the inspector windows in the lower right, and so forth (see Figure 3-17). I encourage you to experiment with these options on your own.

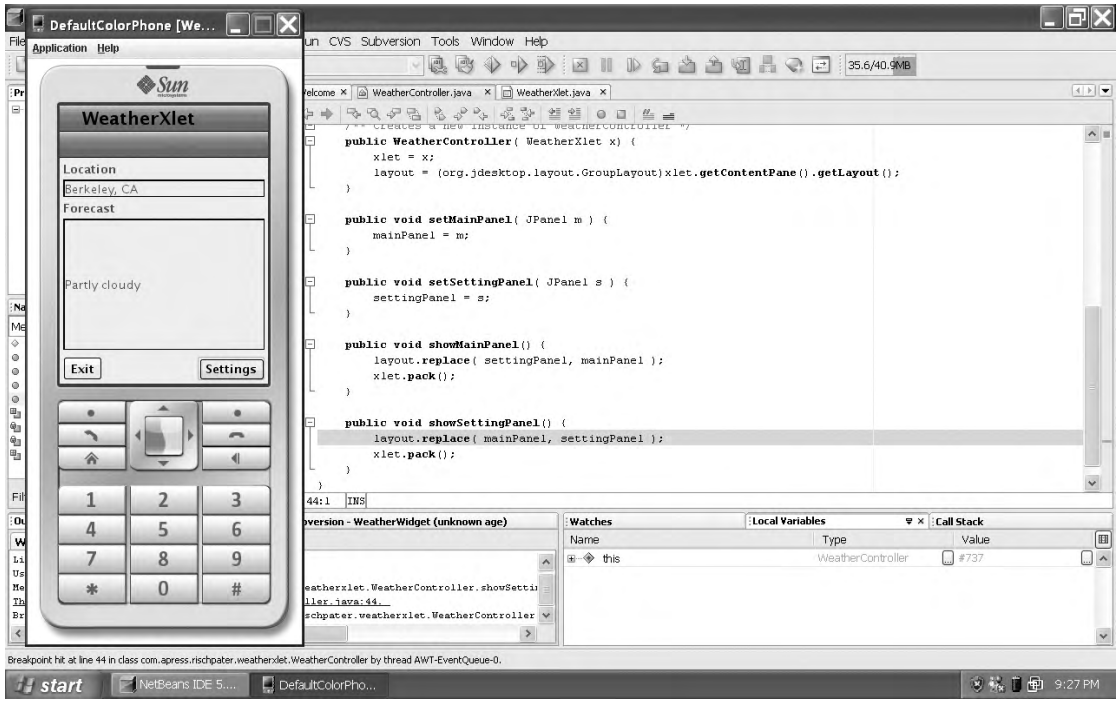


Figure 3-17. *Execution paused at a breakpoint for debugging*

Packaging and Executing CDC Applications

How you package and execute your Java ME applications on CDC-enabled devices will vary from device to device, but Java Web Start and the Java Network Launching Protocol (JNLP) let you work the same way you do with desktop applications. Like packaging for the CLDC/MIDP, using one of these means for packaging your application involves creating both a JAR file for your application and an accompanying descriptor file, as well as an additional policy file that indicates the permissions required by the application. Unlike

the CLDC/MIDP, the descriptor is an XML file with three main sections: a `<resources>` section specifying the required runtime and the location of the JAR file for the application, an `<information>` section indicating the title and vendor of the application, and an `<application-desc>` entry indicating the main class of the application. All are wrapped in a `<jnlp>` XML tag. Listing 3-13 shows an example.

Listing 3-13. *The Descriptor File for WeatherApplet*

```
<?xml version="1.0" encoding="utf-8" ?>
<jnlp codebase="sb:///WeatherApplet/">
  <resources>
    <Java SE version="1.4+"/>
    <jar href="lib/classes.jar"/>
  </resources>
  <information>
    <title>WeatherApplet</title>
    <vendor>Ray Rischpater</vendor>
  </information>
  <application-desc main-class="com.apress.rischpater.weatherxlet.Weather➥
Xlet">
  </application-desc>
</jnlp>
```

You can write this by hand, or you can have the NetBeans IDE roll a deployment for you, much as it would a CDC/MIDP deployment. When it does this, it creates a deployment image suitable for a SavaJe device, with the following directory hierarchy:

```
bundle.jnlp
bundle.policy
lib/
lib/classes.jar
```

The `bundle.jnlp` file is the descriptor you see in Listing 3-13; the `bundle.policy` file requests the permissions for the application, like this:

```
grant codeBase "sb:/WeatherApplet/lib/classes.jar" {
  permission java.security.AllPermission;
};
```

This example requests all permissions; in practice, you may want to apply the principle of least privilege, which requires that *only* the permissions required by the applications be listed in the `grant` block.

■ **Note** The `codeBase` attribute in the permissions block should match the `codebase` attribute in the JNLP file. Note the difference in capitalization, too!

To use NetBeans to create a suitable deployment image, make sure you have selected a destination path for the image in the project's properties (under the Deployment tab), such as the path to a Secure Digital (SD) card for the target hardware. Then simply right-click the project and choose Deploy Target Bundle. When the build and deployment are complete, you can transfer the media to the target hardware and execute your application.

Wrapping Up

Various tools are available for developing Java ME applications, but if you're just starting out, one of the best is the NetBeans IDE for Java ME. Through the addition of two optional packages—the NetBeans Mobility Pack for CLDC and the NetBeans Mobility Pack for CDC—you can write Java ME applications on Linux or Windows platforms (or, through virtualization, on other Intel-based platforms as well). The IDE provides a visual editor for developing your application GUI as well as integrated build tools to compile and build your CLDC or CDC applications.

CLDC/MIDP applications are called MIDlets and have a well-defined life cycle that permits you to pause applications at any point during execution. CDC applications also have the same life cycle and are called Xlets. While you can use a subset of Java Swing for many CDC-based applications, the same is *not* true of the CLDC/MIDP, which requires you to use a GUI and event hierarchy developed expressly for constrained mobile devices.

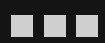
While both the CLDC and the CDC use the same basic build process and build tool chain, later steps of the build and packaging process diverge between the two environments. Both use the JDK's compiler, and you should be sure to use an obfuscator when generating production code for either the CLDC or the CDC to save space. However, the CLDC/MIDP splits bytecode verification into two phases, so CLDC/MIDP build environments must preverify the compiled byte code after obfuscation before packaging. Finally, the mechanics of packaging differ between the two platforms; CLDC/MIDP devices use a combination of a flat text file that describes the application and a JAR file, while CDC applications generally use the same JNLP (consisting of an XML file and a class file) mechanism used by desktop applications.

Despite the differences, the NetBeans IDE provides a unified visual development environment with consistent tools for editing, debugging, building, and packaging Java ME applications.

Intermezzo

Until now, you've been learning generalities about Java ME and the two configurations that comprise it, focusing largely on the underlying profiles. In the next two parts of the book, I turn your attention to the key libraries that support application development. Part 2 focuses on the MIDP for the CLDC, and Part 3 discusses the Foundation and Personal Profiles for the CDC. Where you go from here depends largely on your interests. If you'd like to learn the most about Java ME as a platform, then start with Part 2 and read through Part 3, recognizing that what's discussed is complementary, yet few devices on the market offer *all* the capabilities you're learning about. If you're interested in targeting a particular configuration, read just Part 2 or Part 3 for the configuration and profiles that interest you.

PART 2



CLDC Development with MIDP

For the average developer, Java ME is synonymous with the MIDP, and for good reason. For more than half a decade, developers have been successfully delivering games, entertainment, productivity, and other applications atop the MIDP—the first commercial success for mobile Java in the marketplace. In this part, you learn what it takes to build successful MIDP applications.

CHAPTER 4

Introducing MIDlets

The MIDlet is at the heart of the Java ME CLDC execution model. MIDlets are the equivalent of applications in the CLDC/MIDP domain, processing input from the user and presenting output. MIDlets also manage interactions with the system, responding to requests for resources by pausing and yielding control of the system to the interrupting application.

In this chapter, I discuss the MIDlet interface that you must implement within your application. I begin by showing you the source code for the simplest MIDlet, and I discuss the methods you must implement to satisfy the MIDlet interface. I discuss the life cycle of MIDlets, and I give you more information on packaging MIDlets. Next, I show you how a MIDlet loads properties and resources from its accompanying JAR file. Finally, I show you the various ways you can start MIDlets.

Looking at the Simplest MIDlet

Listing 4-1 shows a simple Hello World MIDlet, which implements the methods required of all MIDlets. This MIDlet presents a single (editable) text box with the message "Hello World."

Listing 4-1. *The Hello World MIDlet*

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class HelloMidlet extends MIDlet {
    public HelloMidlet() {
    }

    public void startApp() {
        Display.getDisplay(this).setCurrent(
            new TextBox("", "Hello World", 20, 0) );
    }
```

```
    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

Figure 4-1 shows the Hello World MIDlet and its message.

Figure 4-1. *The Hello World MIDlet's interface*

All MIDlets must implement four methods:

- `HelloMidlet` (the MIDlet constructor): The system invokes this method when it needs to construct an instance of the MIDlet. The constructor typically does nothing, deferring initialization to the `startApp` method.
- `startApp`: The Application Management Software (AMS) invokes this method when your application is launched or resumed. `startApp` should perform any necessary bootstrapping to initialize your application and present the application's first screen. Simple applications may choose to do this within the confines of this method; more sophisticated applications may chain to a separate method.
- `pauseApp`: The AMS invokes this method when the system must interrupt the MIDlet for any reason, such as an incoming call or message, or the launch of another application. Your MIDlet should release any unneeded resources at this point.
- `destroyApp`: The AMS invokes this method when the application must exit, either as a result of user or system input.

■ **Tip** Technically, a MIDlet doesn't need to have a constructor; if you don't supply one, Java will create one for you. It's a good practice to declare one, however.

On creation, this MIDlet sets a new text box—which implements the `Displayable` interface—as the current display, and it returns control to the MIDlet GUI framework. Because it's so simple, the MIDlet does nothing when the application is paused, and it relies on the native garbage collection when the application exits.

Understanding the MIDlet Life Cycle

I introduced the MIDlet life cycle in Chapter 3, so Figure 4-2 should look familiar to you. At any time, a MIDlet may be in one of three states:

- *Paused*: A paused application does not receive events from the system and is awaiting activation or destruction.
- *Active*: An active application receives events from the system and can interact with the user.
- *Destroyed*: A destroyed application has completed its execution and is awaiting garbage collection by the virtual machine's garbage collector.

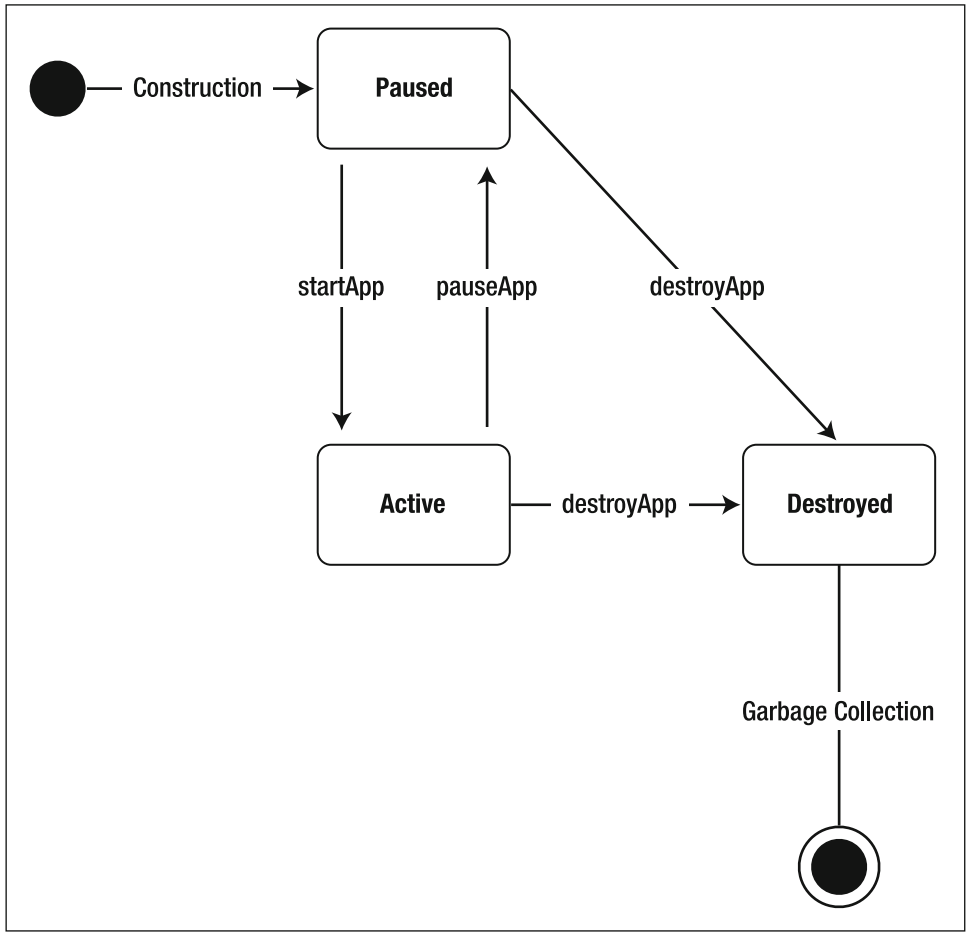


Figure 4-2. *The MIDlet application life cycle*

When the AMS creates a MIDlet, the MIDlet begins its life in the paused state. Only once the AMS invokes the MIDlet's `startApp` method does the MIDlet enter the active state; at this point, you can interact with it. This method isn't just the MIDlet's entry point, however; the AMS invokes `startApp` whenever the MIDlet is about to enter the active state.

At any point, the AMS can interrupt the MIDlet, forcing it to again enter the paused state. The most common example of this is on wireless terminals when an incoming call is received; of course, it also may happen on behalf of another event, such as an incoming message, insufficient resources to perform a system task, and so on. Regardless of the cause, the AMS invokes your MIDlet's `pauseApp` method. The `pauseApp` method should release any shared resources and leave your MIDlet in a quiescent state, because the AMS or device operating system will be bringing another application to the foreground.

At any point, the AMS may signal to your MIDlet that it should terminate by invoking its `destroyApp` method. When the AMS invokes `destroyApp`, the AMS passes a flag indicating the MIDlet has a choice to exit or not to exit. When this flag is `false`, your application can signal the AMS not to exit by throwing a `MIDletStateChangeException`. It's up to the platform provider to determine under what conditions a MIDlet exit is unconditional, so your MIDlet design shouldn't rely on being able to control when it terminates.

Of course, you can trigger these state changes programmatically, too. If your MIDlet wants to relinquish control to the AMS and enter the paused state, simply invoke `notifyPaused`. In turn, the AMS will place your MIDlet in the paused state, although it will do so *without* invoking your `pauseApp` method, because it will assume that you were prepared to pause execution before you invoked `notifyPaused`. If you need to return to the active state from the paused state, invoke `resumeRequest`; the AMS will process your request to resume and invoke your `startApp` method when it's ready to return your MIDlet to the active state. In a similar vein, you can invoke `notifyDestroyed` to exit your MIDlet; like `notifyPaused`, this does not signal your MIDlet through the `destroyApp` method, but instead immediately terminates your MIDlet.

■ **Caution** Do not invoke `System.exit` from your MIDlet! The *only* way to exit your MIDlet is by calling `notifyDestroyed`, which signals to the AMS that you wish to exit. Invoking `System.exit` from a MIDlet generates a `SecurityException`.

Packaging MIDlets

As you learned in Chapter 3, devices obtain MIDlets through two files: JAD and JAR. The JAD file contains a summary of the MIDlet's requirements for execution, and devices may download this file to determine the suitability of the MIDlet for execution within the device's Java ME environment. However, only the JAR file is actually required.

Within the JAR file is a *manifest*, named `manfest.mf`. The manifest exists at the root of the JAR file and contains a detailed description of the contents of the JAR file. It consists of name-value pairs, called *MIDlet attributes*, which are separated by colons and delimited by newlines. Table 4-1 shows the attributes defined by the Java ME specifications. In addition to the attributes shown in Table 4-1, you can pass additional attributes to the MIDlet in the manifest; these are available by calling the MIDlet method `getAppProperty` and passing the name of the attribute.

Table 4-1. *MIDlet Attributes*

Attribute	Required?	Purpose
`MicroEdition-Configuration`	Y	The name and version of the Java ME configuration required by the MIDlet
`MicroEdition-Profile`	Y	The name and version of the MIDP required by the MIDlet
`MIDlet-Data-Size`	N	The minimum number of bytes of persistent storage used by the MIDlet suite
`MIDlet-Delete-Confirm`	N	A message to display when the AMS confirms the deletion of the MIDlet suite
`MIDlet-Delete-Notify`	N	A URL to notify when the MIDlet suite is deleted
`MIDlet-Description`	N	A textual description of the MIDlet suite
`MIDlet-Icon`	N	The absolute path of a Portable Network Graphics (PNG) file in the JAR file that represents the MIDlet suite
`MIDlet-Info-URL`	N	A URL that points to further information about the MIDlet suite
`MIDlet-Install-Notify`	N	A URL to notify when the MIDlet suite is installed
`MIDlet-Jar-Size`	N	The size (in bytes) of the MIDlet suite's JAR file
`MIDlet-Jar-URL`	N	The URL from which the AMS can obtain the JAR file
`MIDlet-`*n*	Y	The name, the path to the icon, and the class of the *n*th MIDlet in the suite
`MIDlet-Name`	Y	The name of the MIDlet suite
`MIDlet-Permissions`	N	A list of permissions required by the MIDlet suite
`MIDlet-Permissions-Opt`	N	A list of permissions that are used but not critical to the operation of the MIDlet suite
`MIDlet-Push-`*n*	N	A list of push registry entries to cause this MIDlet to autostart
`MIDlet-Vendor`	Y	The name of the organization that provides the MIDlet suite
`MIDlet-Version`	Y	The version number of the MIDlet suite

As the table indicates, the AMS *requires* six of these attributes in order for the AMS to install your application:

- `MIDlet-Name`: Specifies the name of your MIDlet suite as a catalog entry when displaying memory use and so forth. The AMS may present this name to the user during installation.

- `MIDlet-Version`: Specifies the version of your MIDlet suite, which is used to determine the currently installed version when upgrading a MIDlet suite. The AMS may present the version to the user. The version must be in the form *major-version.minor-version.micro-version*, where *major-version, minor-version*, and *micro-version* are all integers. (The *micro-version* value is optional.)

- `MIDlet-Vendor`: Specifies your institution's name, which is the publisher of the MIDlet suite.

- `MIDlet-`*n*: Specifies the name, icon, and entry class (delimited by commas) for the *n*th MIDlet in the suite. There must be at least one of these (`MIDlet-1`) to specify a single MIDlet in the suite.

- `MicroEdition-Configuration`: Specifies the version of the Java ME configuration (for example, CLDC 1.0) required by the suite.

- `MicroEdition-Profile`: Specifies the version of the Java ME profile (for example, MIDP 2.0) required by the suite.

In addition to the manifest, which describes the JAR file to the AMS, the AMS can use the JAD file to determine the suitability of the MIDlet suite for the target device prior to downloading and installing the suite itself.

■ **Caution** Most platforms require you to provide a JAD file as well as a JAR file. If you do so, make sure that the entries in your JAD file match the manifest *exactly*, or else the AMS may not install your application.

Obtaining Properties and Resources

The MIDlet properties from the manifest and JAD file are available to your application at runtime. The MIDlet method `getAppProperty` takes the name of a property to return, and returns the value of the property if it is set, or `null` if it's not. Note that attribute names are case sensitive, so `MyProperty` is *not* the same as `myProperty` or `Myproperty`. When searching for a property, `getAppProperty` follows these rules:

- If the MIDlet suite's JAR file is cryptographically signed, `getAppProperty` searches the manifest first and then searches the JAD file. If it finds values in both files, it knows they must match.

- If the MIDlet suite's JAR file is not signed, `getAppProperty` searches the JAD file first. It searches the manifest for the property only if it doesn't find an entry for the property in the JAD file.

Of course, JAR files can contain more than just classes. You can obtain a named resource in your JAR file using `java.lang.Class.getResourceAsStream`, which returns an `InputStream` to the file in JAR file. The `javax.microedition.lcdui.Image` class, which is responsible for drawing bitmap images in PNG format, also includes the `createImage` class method, which takes a path to an image and returns an `Image` instance that's ready to display.

Managing Startup Events and Alarms

In addition to user input, external events such as incoming messages and alarms can trigger the activation of your MIDlet. To use an external event to trigger the activation of your MIDlet, your MIDlet must register with the AMS to indicate that it wants to receive push events, and it must have the privilege to do so.

When the MIDlet is not running, the AMS and the MIDlet share responsibility for handling an incoming event, whether it's an alarm or an incoming connection request (see Figure 4-3). The AMS monitors registered push events on behalf of inactive MIDlets, and it starts MIDlets in response to incoming events. On the other hand, when the MIDlet is running, the push directory sends the push event directly to the MIDlet.

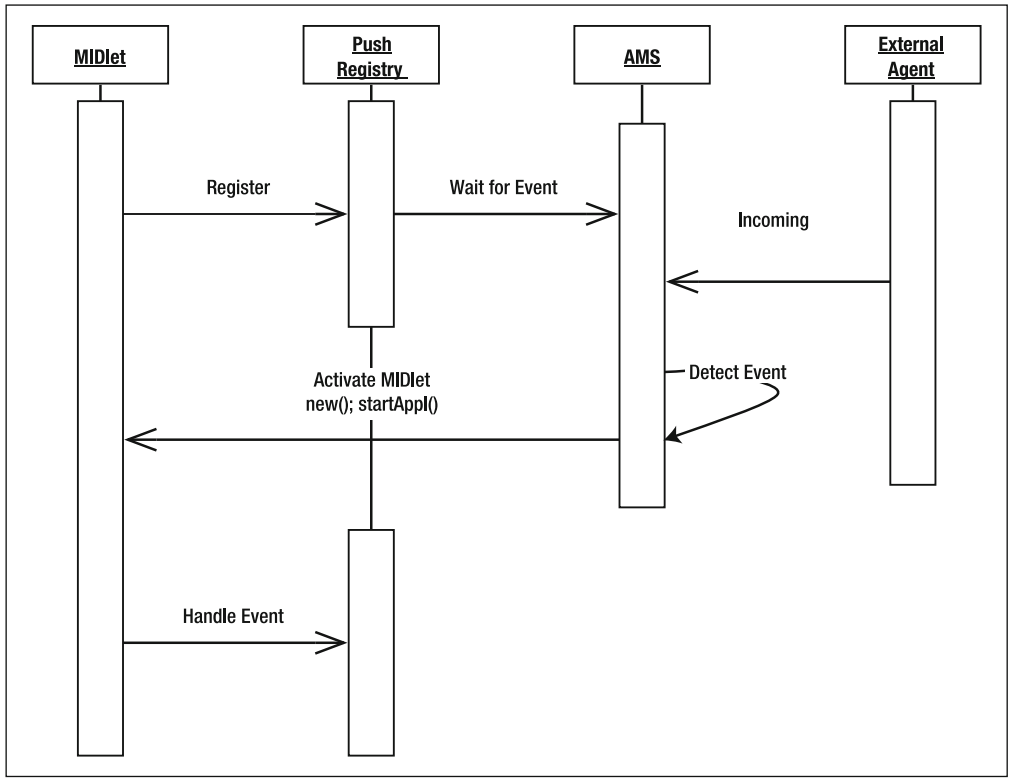


Figure 4-3. *The relationship between the AMS and the MIDlets awaiting startup events*

I save the details of how to use the push registry for incoming messages until Chapter 14 when I discuss the Wireless Messaging API in detail. For now, you just need to know that the MIDlet alarm mechanism also operates through the same mechanism, with one exception: MIDlets can only register for alarms through the push registry at runtime, not via the manifest.

To register for an alarm, you use the `PushRegistry.registerAlarm` method, which is available from the `javax.microedition.io` package. This method takes the name of a MIDlet to which the alarm should be sent, along with the time at which the alarm should fire. A MIDlet may have at most one active alarm; the `registerAlarm` method returns the last scheduled launch time; to cancel the alarm, pass `0` for the time.

You should only use an alarm when an application is not running. When an application is running, you should use the `Timer` and `TimerTask` classes provided by the MIDP instead. The `java.util.Timer` class manages a single `java.util.TimerTask` subclass; to use it, create a subclass of `java.util.TimerTask` that overrides its `run` method with the code that must be performed. Then schedule the subclass's execution using the timer's `schedule` method. Listing 4-2 shows the relationship between alarms and timers for a simple MIDlet that notifies the user 15 seconds after it is launched, even if you terminate the MIDlet before the notification appears.

Listing 4-2. *Using Alarms and Timers Together*

```
package com.apress.rischpater.alarmtimer;

import java.io.*;
import java.util.*;
import java.lang.*;
import javax.microedition.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;

public class AlarmTimerMidlet extends MIDlet implements CommandListener {
    private Form infoForm;
    private StringItem helloStringItem;
    private Command exitCommand;
    private Alert alarmAlert;
    private long DELAY = 15 * 1000;
    private Timer timer;
    private MyTask task;
    private long whenLaunched = new Date().getTime();
    private String storeName = "AlarmTimerStore";
    private RecordStore store;
```

```
public AlarmTimerMidlet() {
}

class MyTask extends TimerTask {
    private AlarmTimerMidlet owner;
    void setOwner( AlarmTimerMidlet o ) {
        owner = o;
    }
    public void run() {
        owner.alarmFired();
    }
}

private void initialize() {
    try {
        Date d = new Date();
        long whenToFire = d.getTime() + DELAY;
        store = RecordStore.openRecordStore(storeName, true);

        if (store.getNumRecords()>0){
            byte b[] = store.getRecord(1);
            ByteArrayInputStream bais = new ByteArrayInputStream(b);
            DataInputStream dis = new DataInputStream(bais);
            whenToFire = dis.readLong();
            if (whenToFire < whenLaunched) {
                alarmFired();
            }
            store.deleteRecord(1);
            store.closeRecordStore();
            return;
        }
        store.closeRecordStore();

        String me = this.getClass().getName();
        PushRegistry.registerAlarm(me, 0);
        timer = new Timer();
        task = new MyTask();
        task.setOwner(this);
        timer.schedule(task, whenToFire - d.getTime());

        getDisplay().setCurrent(get_infoForm());
    }
    catch( Exception e) {};
}
```

```
public void commandAction(Command command, Displayable displayable) {
    if (displayable == infoForm) {
        if (command == exitCommand) {
            exitMIDlet();
        }
    }
}

private void scheduleMIDlet( ) {
    try {
        String me = this.getClass().getName();
        Date when = new Date();
        if ( when.getTime() < whenLaunched + DELAY ) {
            PushRegistry.registerAlarm(me, whenLaunched + DELAY);
        }

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream( baos );
        dos.writeLong( whenLaunched + DELAY );
        byte b[] = baos.toByteArray();
        store = RecordStore.openRecordStore(storeName, true);
        store.addRecord(b, 0, b.length);
        store.closeRecordStore();
    }
    catch (Exception e) {}
}

private void alarmFired() {
    getDisplay().setCurrent(get_alarmAlert(), get_infoForm());
}

public Display getDisplay() {
    return Display.getDisplay(this);
}

public void exitMIDlet() {
    getDisplay().setCurrent(null);
    try {
        if ( getDisplay().getCurrent() != get_infoForm() )
        {
            scheduleMIDlet();
        }
    }
```

```
        catch(Exception e) {};
        destroyApp(true);
        notifyDestroyed();
    }

    public Form get_infoForm() {
        if (infoForm == null) {
            infoForm = new Form(null, new Item[] {get_helloStringItem()});

            infoForm.addCommand(get_exitCommand());
            infoForm.setCommandListener(this);
        }
        return infoForm;
    }

    public StringItem get_helloStringItem() {
        if (helloStringItem == null) {
            helloStringItem = new StringItem("",
                "An alarm has been set for fifteen seconds from now.");
        }
        return helloStringItem;
    }

    public Command get_exitCommand() {
        if (exitCommand == null) {
            exitCommand = new Command("Exit", Command.EXIT, 1);
        }
        return exitCommand;
    }

    public Alert get_alarmAlert() {
        if (alarmAlert == null) {
            alarmAlert = new Alert(null, "The alarm has fired.\n", null, null);
            alarmAlert.setTimeout(-2);
        }
        return alarmAlert;
    }

    public void startApp() {
        initialize();
    }
```

```
    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

The code begins with a `TimerTask` subclass that notifies the MIDlet that the alarm fired; this subclass needs two methods: `setOwner`, which hooks the task to the MIDlet, and `run`, which performs the action when the timer invokes the task.

The `initialize` method, which `startApp` invokes when the AMS launches the MIDlet for any reason, must do two things. First, it must determine whether the MIDlet is launching because of an alarm invocation or a user invocation, and second, it must display the appropriate screen (either an informative form or the alarm alert) depending on whether the alarm fired while the application was not running. To do this, the MIDlet must persist the desired alarm time so that it can tell the difference between a user-initiated launch and an alarm launch. To persist this data, the MIDlet uses the Java ME record store, which I discuss in more detail in Chapter 6. For now, it suffices to know that the record store persists records that are arrays of bytes; I use instances of the `ByteArrayInputStream` and `DataInputStream` classes to decode the desired alarm time if the MIDlet detects a record in the store. (The `scheduleMIDlet` method, which I discuss next, is responsible for writing this record upon exit if an alarm is set.) If the alarm time was before the current time, the MIDlet will show the alarm alert by setting it to the current displayable item. If no record was found, or if the alarm time hasn't been reached, the MIDlet will create a new instance of `MyTask` and schedule its task to fire at the appropriate time.

The `scheduleMIDlet` method does the opposite. Invoked upon the MIDlet exit, it checks the current time against the launch time, and if it is before the time for the alarm to fire, it will register the alarm with the push registry before writing the alarm time to the persistent record store using instances of `ByteArrayOutputStream` and `DataOutputStream`. The `exitMIDlet` method invokes `scheduleMIDlet` when you exit the application through the exit command (so the alarm is only scheduled on normal exits, not on AMS-induced termination).

Both the timer and the alarm use the `alarmFired` method, which simply sets the current display to be the alarm alert notification instead of the default informative screen the MIDlet shows at application launch.

The remainder of the code initializes the user interface:

- `get_infoForm`: This method lazily creates an instance of the form shown at application launch.

- `get_helloStringItem`: This method lazily creates an instance of the string item used by the introductory form.

- `get_exitCommand`: This method lazily creates an instance of the exit command used by the MIDlet to signal application exit.
- `get_alarmAlert`: This method lazily creates the alarm alert message.

Wrapping Up

The `javax.microedition.midlet` class encapsulates the notion of an MIDP application, called a MIDlet. To implement a MIDlet, you must subclass this class and provide four functions: the constructor, `startApp`, `pauseApp`, and `destroyApp`. The AMS invokes these methods as your MIDlet transitions from paused to active to destroyed (potentially entering the paused and active states multiple times as the native platform interrupts your application). You can cause one of these transitions; for example, you can force your MIDlet to exit by invoking `notifyDestroy` after cleaning up the resources used by your MIDlet.

MIDlets are packaged as suites contained within a JAR file that contains both the classes implementing the MIDlets for a suite and a manifest that describes the suite through properties such as the MIDlet suite name, the icon, and individual MIDlet names, icons, and MIDlet subclasses. The JAR file is usually accompanied by a second file—the descriptor (JAD) file—which contains the same properties identified in the manifest file. You can query the property list for a MIDlet using the MIDlet method `getAppProperty`, or you can load a file from the JAR file using the `Class` method `getResourceAsStream`.

Not just user actions invoke MIDlets. MIDlets can also start in response to incoming events such as connection requests or timer events. You can schedule an alarm using the push registry `registerAlarm` method, which specifies the MIDlet to start and when to start. For running MIDlets, you'll want to subclass `TimerTask` and use an instance of the `Timer` class to run your task.

CHAPTER 5

Building User Interfaces

The CLDC brings together a number of flexible user-interface widgets from which you can build many kinds of applications. Unlike other GUI platforms, the interfaces to these UI widgets are highly abstracted, meaning you can write one application to run on a number of different device configurations, such as those with keypads, touchscreens, or even voice input. In fact, it's fair to say that the CLDC's generality has outstripped the imagination of most device manufacturers; the interface to the widgets supports far more kinds of hardware than have been commercially available.

In this day and age of drag-and-drop code generation to build user interfaces, it's tempting to gloss over these details. After all, why worry too much about how forms collect visible widgets when you can simply pick a specific form or widget from a palette and then wire things together by pointing and clicking? Besides the obvious answer—you can better design your application with a firm grasp of the fundamentals—understanding these fundamentals enables you to create your own user-interface widgets, as well as efficiently compose sophisticated applications that remain easy to navigate.

In this chapter, I show you the components you can use to build your application's user interface. I begin by discussing the relationship between the various elements of the `javax.microedition.lcdui` package and showing you how the various elements fit together. I then discuss *commands*—the fundamental way users interact with your application. After that, I give you a comprehensive introduction to the various visible objects you can create, beginning with simple items for showing text and choices, and moving on to how you group these items into application screens. Finally, I discuss how these items interact with the display canvas so that you can understand how to make your own visible objects.

Understanding the Relationship Between the Display and Visible Item Objects

The ultimate purpose of MIDlets is to interact with the user. In a very real sense, your MIDlet's flow can be reduced to the following process: pick a `Displayable` to show, set the `Display` to show the `Displayable`, wait for a `Command`, and then pick the next `Displayable` (or exit). The state machine diagram in Figure 5-1 shows this process.

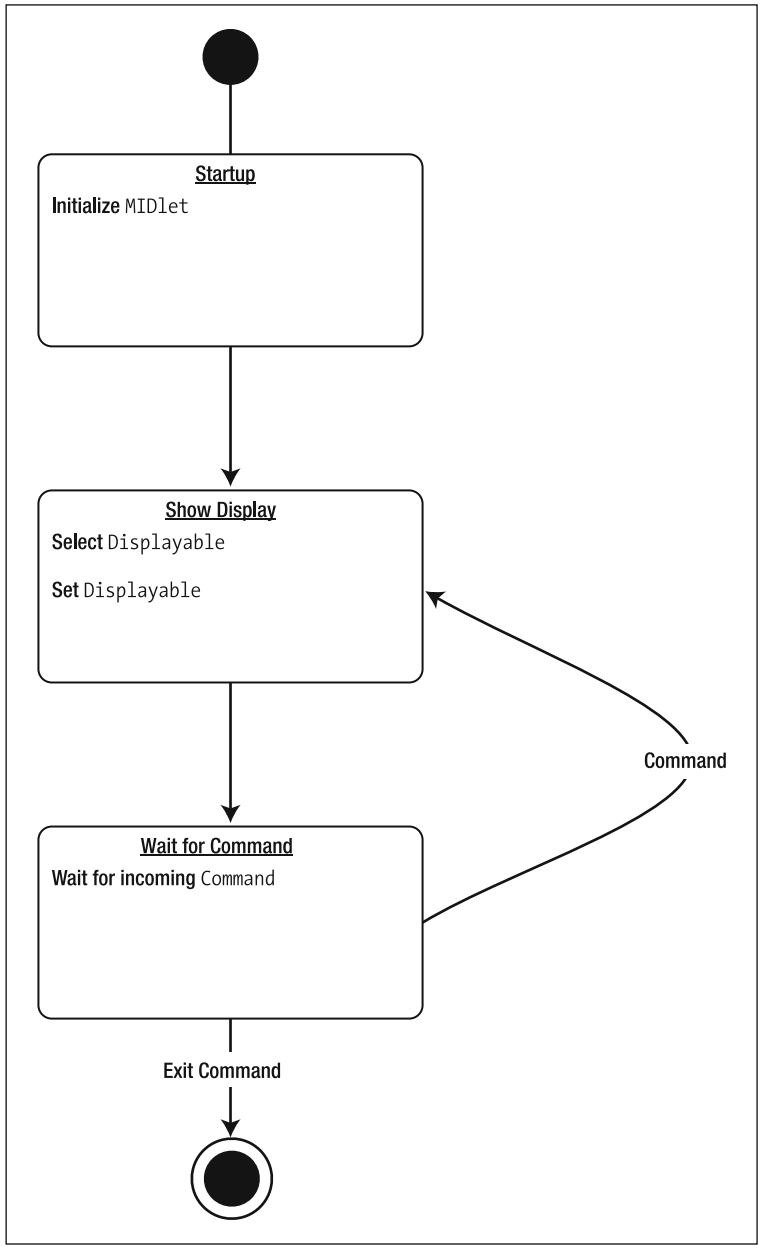


Figure 5-1. *The life cycle of a MIDlet from the perspective of the display*

To facilitate this process, every MIDlet has access to the display, which is represented by an instance of the `Display` class. Obtained through the static method `Display.getDisplay()`, the instance lets you do the following:

- Determine whether the screen supports color or grayscale
- Determine the number of colors and alpha levels supported by the display
- Get the border style and user-selectable color for the foreground and background
- Flash the display backlight
- Obtain the best image bounds for an `Alert`
- Get the currently shown `Displayable`
- Set the currently shown `Displayable`
- Vibrate the device using the vibration motor

What exactly is the `Displayable` interface for? The hierarchy in Figure 5-2 shows the relationship between the `Display`, what it can display (`Displayable` classes), and `Item` classes, which `Displayable`s contain to make up complex user interfaces.

As you can see in Figure 5-2, the `Display` must have a corresponding `Displayable` object to display to the user. `Displayable` objects come in several flavors. The `Canvas` class provides the lowest level of access to graphics, permitting you to intercept raw events and draw directly to the screen. Its subclass, `GameCanvas`, provides some simple abstractions to facilitate porting applications to a variety of devices. For a more high-level approach to the user-interface layout, there is the `Screen` subclass and its subclasses `Form`, `Alert`, `TextBox`, and `List`. The last three are high-level abstractions, handling all of the layout and event handling, while `Form` lets you group one or more `Item` objects—things such as text and date fields, for example.

If you're familiar with either Java AWT or Swing, it's important to realize that the user-interface model for MIDlets is very different. Two key differences affect how you design your user interface from the outset. First, you have no real control over the layout of your application. Unlike the rich Java UI frameworks provided by AWT and Swing, the layout of your user interface is completely controlled by the `Screen` class and its subclasses, which typically have a simplistic layout policy. Second, MIDlets have no implementation of the model-view-controller (MVC) paradigm, so if you're used to using MVC, you will need to implement it yourself.

The architects of the MIDP imposed these limitations to permit MIDlets to run on the widest possible variety of devices and to ease application porting between devices without requiring a heavyweight all-Java user interface such as that provided by Swing. Unfortunately, these limitations do come with a downside: it's nearly impossible to create a truly bespoke user interface. Different devices generally present the user with differing user interfaces; these changes may be small or may grossly affect the appearance of your application. If you're looking to create a UI with a very specific control appearance and placement, you'll need to code that from the ground up using the `Canvas` or `GameCanvas` classes. I discuss that later in this chapter.

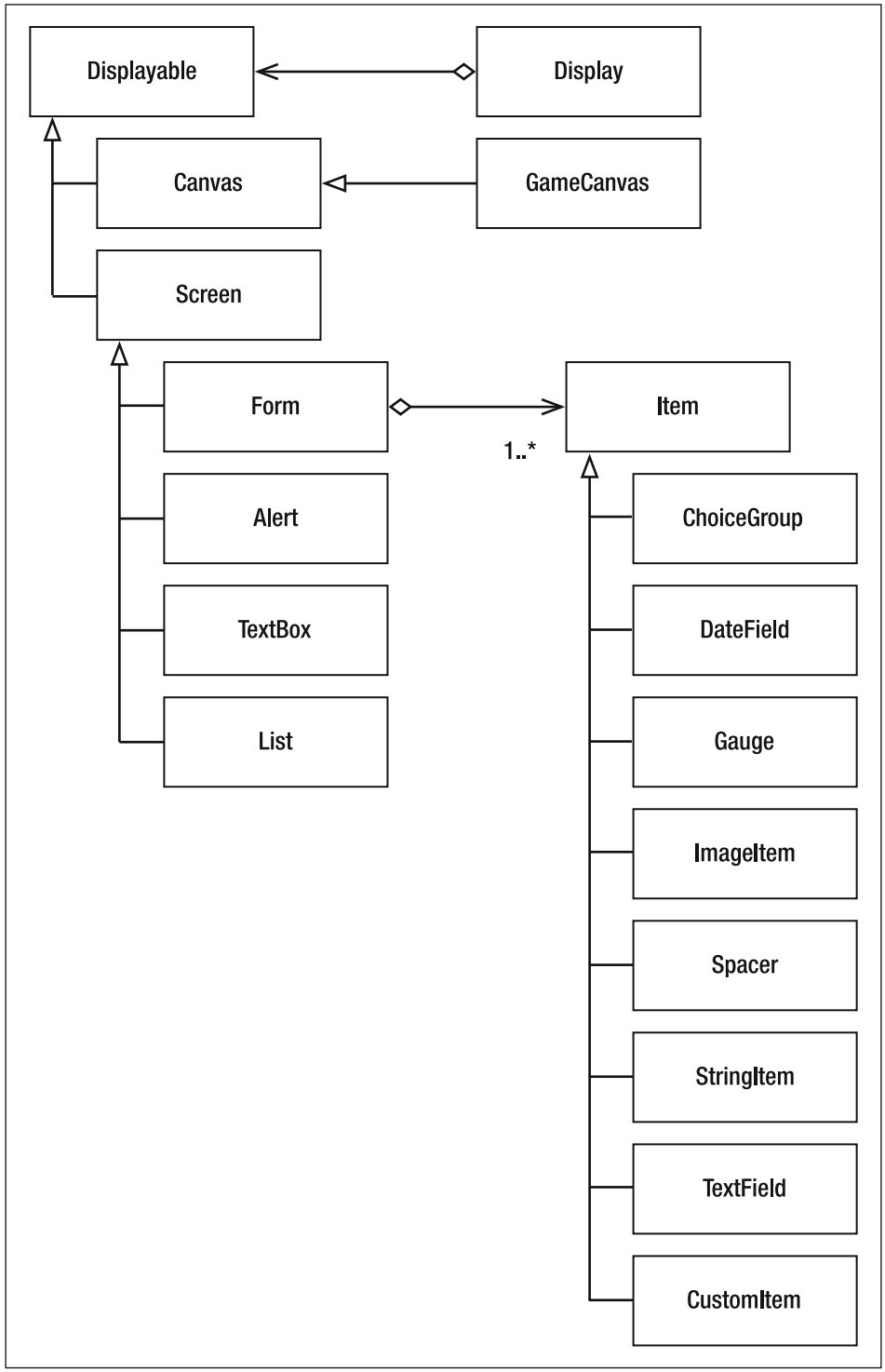


Figure 5-2. *The relationship between the Display, Displayable, and Item classes*

Using Commands to Control Application Flow

Traditional user-interface programming requires implementation based on either event-handling or MVC paradigms (or a combination thereof). The MIDP takes an approach similar to the former paradigm, defining the `Command` class to encapsulate the notion of a command from the user to the application. The MIDP environment sends instances of the `Command` class to registered listeners, much as the Java AWT sends events to event listeners. To receive instances of the `Command` class, MIDP objects must implement the `CommandListener` interface, providing a `commandAction` method. Each `Command` instance is added to a `Displayable` instance, which is responsible for presenting the command to the user in some way (such as via a soft key or menu). The `Displayable` instance is also responsible for sending the instance to a registered listener when you activate the command.

Instances of `Command` are tuples, consisting of a short label, a long label, a type, and a priority:

- *Labels*: Labels specify what the command shows on the user interface; only one label is required. Where the label—and which label—for a command actually appears in the user interface depends on several things, including the priority of the command, the implementation of the `Displayable` to which you add the command, and the implementation of the MIDP runtime itself. (Figure 5-3 shows an example of commands in a soft key menu.)

- *Type*: The type indicates the kind of command, both for your application logic and potentially for the UI of the MIDP runtime. It presents additional information (such as an icon) about the command.

- *Priority*: The priority indicates the relative importance of the command to the user interface. The lower the priority, the more obvious the command's placement on the `Displayable` to which it's assigned.

Figure 5-3. *Three commands—Exit, Help, and Stop—added to a Displayable*

The simplest way for you to understand the relationship between `Command` objects and `Displayable` objects is to show you the code that generated Figure 5-3. It's in Listing 5-1.

Listing 5-1. *Demonstrating the Relationship Between Command and Displayable Objects*

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class CommandExample extends MIDlet {
  public void startApp() {
    Displayable d = new TextBox("Demo", "", 20, TextField.ANY);
    Command exit = new Command("Exit", Command.EXIT, 0);
    Command help = new Command("Help", Command.HELP, 1);
    Command stop = new Command("Stop", Command.STOP, 2);

    d.addCommand(exit);
    d.addCommand(help);
    d.addCommand(stop);

    d.setCommandListener(new CommandListener() {
      public void commandAction(Command c, Displayable s) {
        notifyDestroyed();
      }
    } );

    Display.getDisplay(this).setCurrent(d);
  }
  public void pauseApp() { }
  public void destroyApp(boolean unconditional) { }
}
```

The diagram in Figure 5-4 shows the class relationship for this application. At MIDlet startup, the MIDlet creates a `TextBox` instance and the command instances. Next, it adds each of the new commands to the `TextBox`; in turn, the `TextBox` instance determines where and how the commands should appear based on the labels and priorities. The MIDlet also creates an anonymous `CommandListener` that responds to *all* of the generated commands. It does this using the `TextBox.setCommandListener` method. Finally, the `Display`'s notion of the current display is set to the `Displayable`, so that the `TextBox` will be displayed.

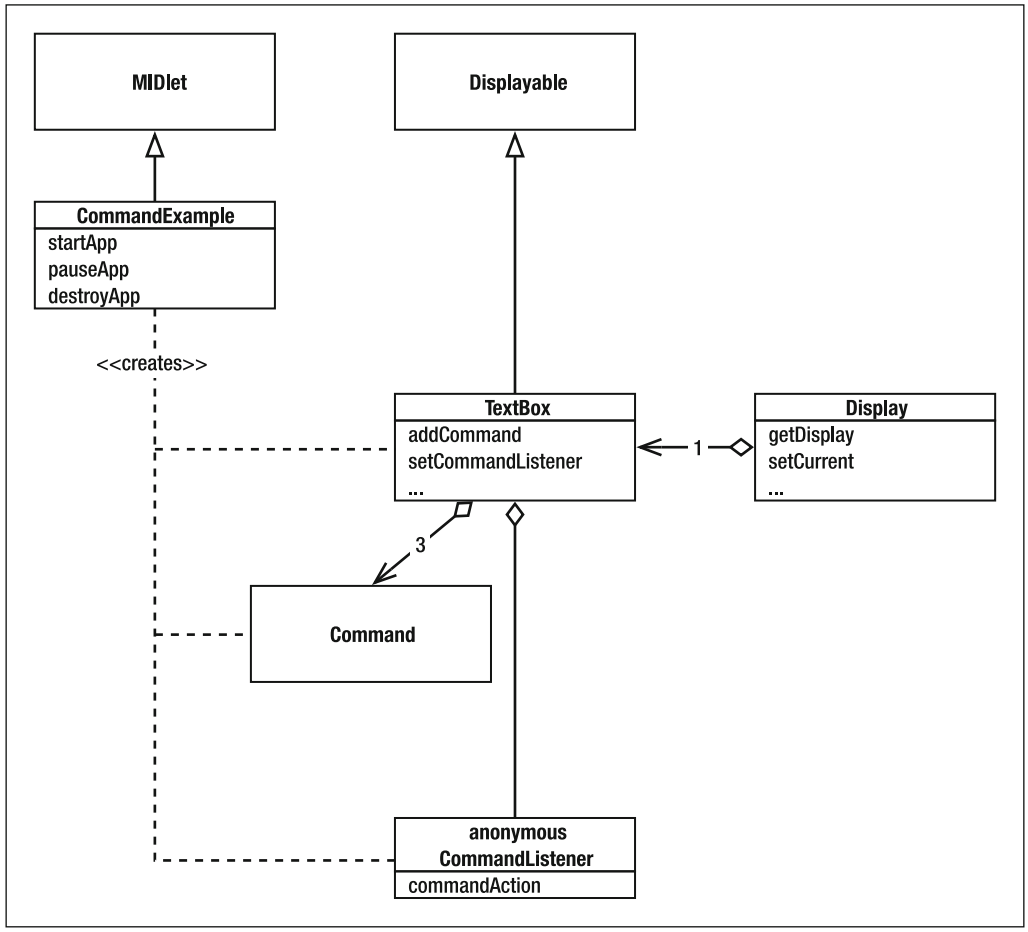


Figure 5-4. *Relationships between a typical Displayable instance, its Command instances, and its CommandListener*

Introducing Basic Visible Items

As noted in a previous section, the `Form` class is responsible for collecting visible items into a single screen. Derived from Sun's UIWidget example, the pseudocode example in Listing 5-2 shows you how to add items to a `Form` at `Form` construction time.

Listing 5-2. *Assembling Some Item Objects on a Form Object*

```
public class StringItemDemo extends MIDlet implements CommandListener {
    private Form mainForm;
    private StringItem stringItem1;
    private StringItem stringItem2;

    public StringItem get_stringItem1() {
         if (stringItem1 == null) {
             stringItem1 = new StringItem(null, "This is a simple label");
         }
         return stringItem1;
     }

     public StringItem get_stringItem2() {
         if (stringItem2 == null) {
             stringItem2 = new StringItem("This is the StringItem label:",
                                          "This is the StringItem text");
             stringItem2.setLayout(Item.LAYOUT_NEWLINE_AFTER | Item.LAYOUT_2);
         }
         return stringItem2;
     }

    public Form get_mainForm() {
        if (mainForm == null) {
            mainForm = new Form("String Item Demo", new Item[] {
                get_stringItem1(),
                get_stringItem2(),
            });
        }
        return mainForm;
    }

    ... more code follows ...
}
```

If you've been playing with NetBeans and creating user interfaces using the GUI builder, this code should look familiar; it's essentially a cleaned-up version of the code built by NetBeans itself. It does, however, demonstrate a common pattern for populating a `Form`: as the `get_mainForm` method shows, you can create the display items at `Form` construction time:

```
mainForm = new Form("String Item Demo", new Item[] {
                get_stringItem1(),
                get_stringItem2(),
            });
```

Passing a list of `Item` instances to the constructor inserts the instances in the same order on the display, and is thus a common way to construct whole screens at a time.

The `Form` class really is a collection, though, and offers several methods to manipulate the items it displays. I show you those in a later section in this chapter, "Collecting Visible Items Using the Form Class." For now, though, you should know that you can also add a single item to the `Form` instance using its `append` and `insert` methods. The `append` method adds the `Item` instance to the end of the list of items on the `Form`, while the `insert` method inserts the item between two other items on the `Form`.

Introducing Items

For your user interface, the rubber hits the road with the `Item` class and its subclass. Unless you're coding raw interface code using the `Canvas` or `GameCanvas` classes (which I discuss later), each visible item in your user interface must implement the `Item` interface or else be a wholly separate `Displayable` object such as an `Alert`, `TextBox`, or `List`. The `Item` class encapsulates the following responsibilities:

- *Command management*: An `Item` can send commands to the `Form`'s command listener in response to user keystrokes such as selections or other actions. In addition, an individual `Item` can have its own command listeners.
- *Drawing*: An `Item` knows how to draw itself.
- *Event handling*: An `Item` knows how to handle raw events such as keystrokes, so your application doesn't have to.
- *Layout preferences*: An `Item` signals to its containing form its preferences about how it should be laid out, and then the `Form` uses its layout policy to present all of the items on the display in a cohesive way.

The `Item` class provides three methods for managing commands:

- `addCommand`: Lets you add a `Command` instance of type `ITEM` to the item. In turn, the MIDP implementation presents this command when the item is active (highlighted).
- `removeCommand`: Removes the indicated command from the item.
- `setItemCommandListener`: Sets a listener for `ITEM` commands on this item.

With the exception of `CustomItem` subclasses—which I discuss in detail in the last section of this chapter—`Item` subclasses take care of their own drawing. You can, however, customize the drawing behavior of some items. For example, most items accept a label, which is a string that precedes the item and is drawn in a read-only fashion. The `Item` class provides the methods `getLabel` and `setLabel` to get and set the label, and it provides constants to suggest button or hyperlink style via the constructor of specific `Item` subclasses such as `StringItem`.

Finally, `Item` instances signal their preferences as to how they should be laid out in the parent `Form` to the parent `Form`. They do this through the layout constant you set via the `setLayout` method. The `Form` class provides layout flags for indicating that an item should be aligned to the top, bottom, left, right, or center of its viewable rectangle, that an item should be shrunk to its minimum bounds or expanded as much as possible, and whether an item should appear on a subsequent line or whether other items should appear on their own line after it. These layout flags all affect the row-based layout policy of the `Form`, which you'll encounter in the section "Collecting Visible Items Using the Form Class" later in this chapter. You can also query an `Item` for its minimum or preferred size using one or more of these four methods: `getMinimumHeight`, `getMinimumWidth`, `getPreferredHeight`, and `getPreferredWidth`. You use these in conjunction with `setPreferredSize`, which indicates the size of the viewable item in which you'd *like* the `Form` to present the item.

When using NetBeans, if you're editing a `Form` in the Screen Design view, you can select any of the supported items by choosing one from the Form Items section of the Palette (whose location defaults to the upper right-hand portion of the display). In fact, if you're using NetBeans, odds are that you won't instantiate any of these classes from your source code directly, but will instead rely on the code-generation facility to write that code for you.

■ **Tip** The properties of `Items` (and `Displayables`) that you can change are all also available in the Properties window of NetBeans. In general, if you're using NetBeans to build your user interface, you'll need to understand the properties available to a particular Item, but you won't necessarily need to know the interface to mutate those properties, because you set the property on the `Item` in NetBeans, and NetBeans' automatic code-generation facility does the rest.

Introducing the Spacer

Because you can't specify pixel positions for user-interface items, the MIDP implementation provides the `Spacer` class. Instances of the `Spacer` class take a minimum size on creation and can be used to place pads between adjacent (horizontally or vertically) items.

Introducing the StringItem

The `StringItem` class provides a read-only control drawing a text value. `StringItem` instances may bear labels and be drawn as buttons or hyperlinks (see Figure 5-5). How a `StringItem` appears depends on three things: the `label`, `contents`, and `appearance` mode flags. The `appearance` mode can be one of the values `StringItem.PLAIN`, `StringItem.HYPERLINK`, or `StringItem.BUTTON`, yielding the results you would expect.

Figure 5-5. *StringItem instances of different appearances*

You can change the label or contents of a `StringItem` instance at any time via the `setLabel` and `setText` methods. You can also change the font of a `StringItem` object through a property that's available via the `setFont` and `getFont` methods.

Introducing the TextField

For user input, the MIDP provides the `TextField` class. `TextField`s bear *input constraints* that can restrict user input in a variety of ways; for example, it may require that users enter only numeric input or phone numbers. The `TextField` class can also format input in a variety of ways and return only what is input; for example, it might show an entered phone number as (831) 555-1212, yet report to the application that the entered number was 8315551212. You set an input constraint using the `setConstraints` method with a flag value such as `EMAILADDR`, `NUMERIC`, `PHONENUMBER`, `URL`, or `DECIMAL`. These can be accompanied by additional flags (combined using the bitwise OR operator `|`) such as `PASSWORD` (which keeps you from seeing the entered text) or `UNEDITABLE`.

The class also supports the notion of *input modes*, which let you provide hints to the implementation regarding how numeric key presses should be mapped to alphanumeric key presses. You set input modes by specifying the string name for an input mode to the method `setInitialInputMode`; values include the name of Unicode character blocks as defined by the Java SE class `java.lang.Character.UnicodeBlock`, which is preceded by the string `UCB_`, as in the following:

- `UCB_BASIC_LATIN` for Latin languages
- `UCB_GREEK` for the Greek language
- `UCB_CYRILLIC` for Cyrillic languages
- `UCB_HEBREW` for Hebrew languages
- `UCB_ARABIC` for Arabic
- `UCB_THAI` for Thai
- `UCB_HIRAGANA` for Japanese Hiragana syllabary
- `UCB_KATAKANA` for Japanese Katakana syllabary
- `UCB_HANGUL_SYLLABLES` for Korean

The MIDP also defines the following specific input modes:

- `MIDP_UPPERCASE_LATIN` for the uppercase characters of Latin languages
- `MIDP_LOWERCASE_LATIN` for the lowercase characters of Latin languages

When creating a `TextField`, you pass to the constructor the label and default text, followed by the maximum number of characters the `TextField` should permit the user to enter, and finally the constraints (bitwise ORed as appropriate).

The interface to the `TextField` class provides several low-level operations that let you interact directly with the contents of a `TextField`, including the following methods:

- `getCaretPosition`: Returns the current input position
- `getChars`: Copies the contents of the `TextField` into the array you provide, starting at position zero
- `getString`: Returns the contents of the `TextField` as a string
- `delete`: Lets you delete a specific number of characters starting at the offset you provide
- `insert`: Inserts the character array or string you provide at the specified location
- `setChars`: Lets you replace partial runs of characters in the `TextField` by specifying new characters, an offset, and a length
- `setString`: Lets you replace the entire contents of the `TextField` with a new string

For most applications, you'll simply set the desired constraints and input mode (quite possibly using the NetBeans Properties pane) when you create the item, and then get the text of the item when you transition to a new `Form`.

Introducing the DateField

The `DateField` class is an editable component for presenting calendar (date and time) information. Using the `DateField`, you can obtain constrained user input for the date, time, or both the date and time by specifying the `DATE`, `TIME`, or `DATE_TIME` input modes. Figure 5-6 shows an example of the `DateField` class in action.

The `DateField` extends `Item` with four methods:

- `getDate`: Returns the date you enter
- `setDate`: Sets the item to the indicated date
- `getInputMode`: Returns the input mode you set for the item
- `setInputMode`: Sets the input mode

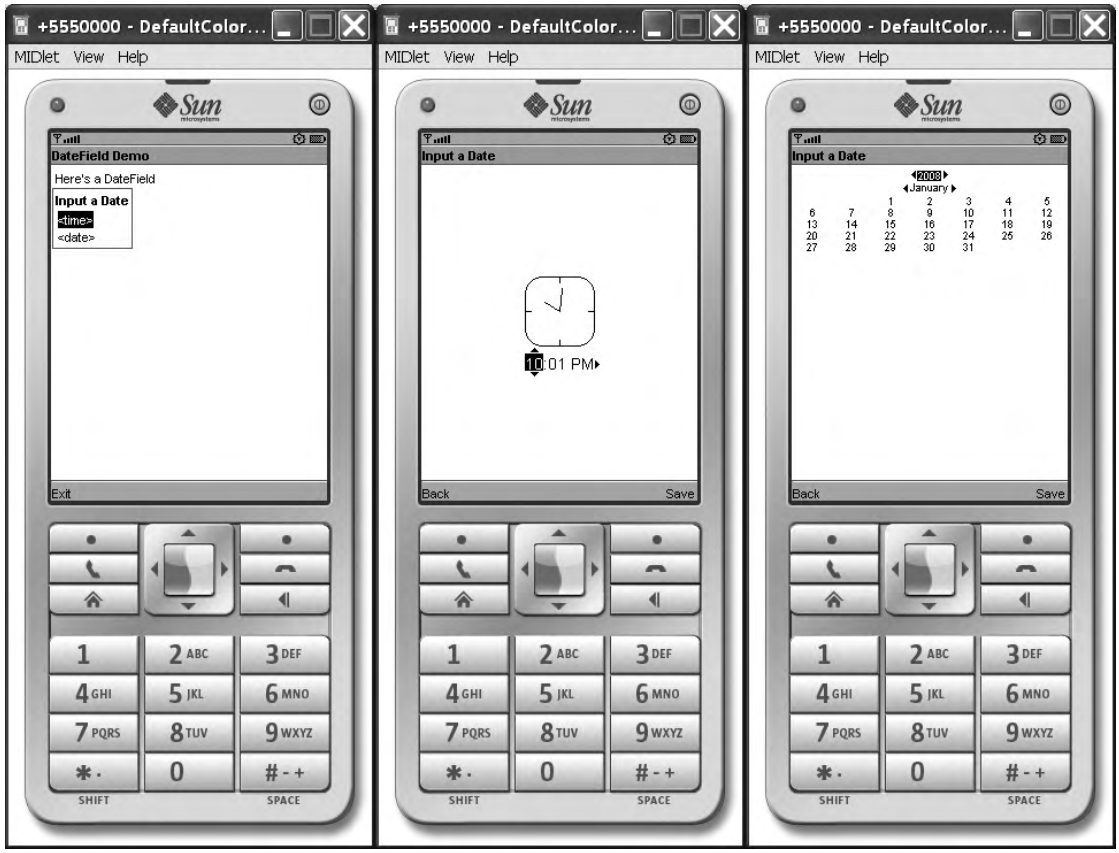


Figure 5-6. *A DateField, and screens showing what happens when you select the time and date portions of the field, respectively*

Introducing the ImageItem

An `ImageItem` shows an image as an image, button, or hyperlink, depending on its appearance mode. As with a `StringItem`, you can provide a `Command` and an `ItemCommandListener` to process selection events on an `ImageItem` appearing as a button or hyperlink.

Introducing the Gauge

A `Gauge` item creates a graphical display of an integer value within a given range between zero and some predefined maximum. When creating a `Gauge` item, you can control the current value and the maximum value. Some instances of `Gauge` are interactive—that is, they let the user set the value. Figure 5-7 shows an interactive `Gauge`.

Figure 5-7. *An interactive Gauge*

Because gauges can be interactive, you can add an `ITEM Command` instance to a gauge, and then the gauge responds when the gauge is changed through its `ItemCommandListener`, just as with an `ImageItem` or `StringItem`.

Managing Choices

The `Choice` interface and `ChoiceGroup` classes let you present a list of choices to users. As shown in Figure 5-8, you can present choices as radio buttons (forcing an exclusive choice, where users can select exactly one item at once), check boxes (where users can select zero or more items), or a pop-up, which shows only the selected item.

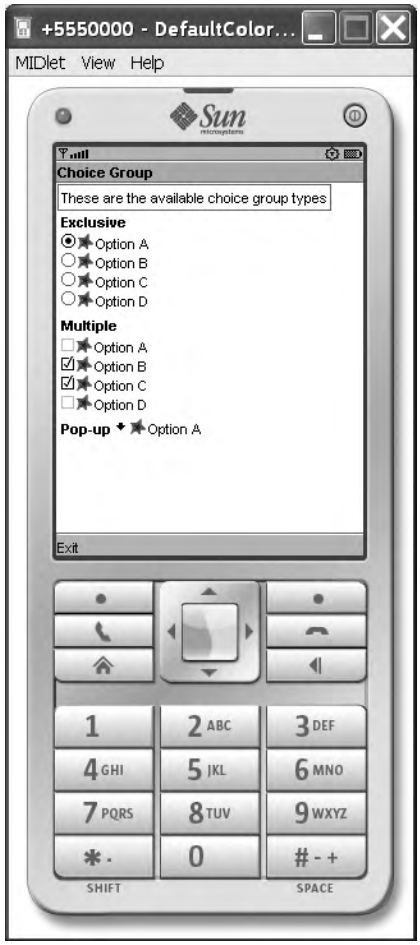


Figure 5-8. *Various kinds of choices presented by the Choice and ChoiceGroup classes*

Because the MIDP provides the `List` class—a class that presents choices—the MIDP standard specifies the `Choice` interface, which is implemented by `ChoiceGroup`, an `Item` subclass, and the `Displayable` subclass `List` (which you'll see in the section "Creating a Custom Item for a Screen" later in this chapter). Generally, you use a `ChoiceGroup` when you want to mix choice items with other user-interface items. On the other hand, a `List` is best if the choices take up the entire display.

When you create a `ChoiceGroup`, you must also specify its type:

- `EXCLUSIVE`: The `ChoiceGroup` can have exactly one element selected at a time, but multiple items may be shown.
- `MULITPLE`: The `ChoiceGroup` can have zero or more items selected at a time.
- `POPUP`: The `ChoiceGroup` can have exactly one element selected, and the selected element is always shown.

You can also specify a list of strings and images, one for each choice the item should show.

The `Choice` interface defines two properties to manage what you have selected from the interface. For exclusive and pop-up lists, the selected index property (with its accessor `getSelectedIndex` and mutator `setSelectedIndex`) is probably the most useful, as it returns the index into the list of choices of the currently selected item. For multiple-selection lists, you want to use `setSelectedFlags` and `getSelectedFlags`, which give you the status of each selected and unselected item through an array of `boolean` flags you provide to the method. You can also invoke `isSelected` with an index to determine if the user has selected a particular item, and `setSelectedIndex` to set the selection status of a particular item.

Because `Choice` manages a collection of user choices, it has a collection-oriented interface with the following methods that let you access and mutate the list of user choices:

- `append`: Lets you append a new user choice (string and image) to the list of choices being presented
- `delete`: Takes an index and deletes the user choice item at the index you specified
- `deleteAll`: Deletes all user choices, resulting in an empty collection of choices
- `insert`: Inserts a new choice item (string and index) after the indicated index
- `size`: Returns the number of user choices in the group

You can also mutate an item's appearance or contents using the `setFont` method to change the font for a specific item.

Introducing the Screen and Its Subclasses

In the beginning of this chapter, I told you how the `Display` class uses `Displayable` subclasses to manage what to display. For high-level user-interface programming, the `Screen` subclasses `Form`, `Alert`, `TextBox`, and `List` are the only game in town. These let you put together complex user interfaces quickly, albeit sacrificing some of the control (such as per-pixel placement) of the `Canvas` class.

Collecting Visible Items Using the Form Class

The most flexible subclass of `Screen` is the `Form` class. As I've already said, it acts as a collection and layout class for `Item` instances, letting you combine various `Item` subclass instances to create screens with a variety of user-interface controls.

The Form class's layout policy centers around rows, by default positioning each Item next to the previous as long as multiple Items fit on a single line. If there are more Items than fit on a single line, Items are placed on rows one below the other. In the unlikely event that there are more Items than fit on the display, the MIDP implementation may choose to provide a scrolling view of the Items on the Form, or may paginate the Form, taking you to a new screen to view additional Items on the Form.

The layout algorithm provides default layout constraints for each Item added to a Form. For example, if the Form is laying out Items left to right, an Item with an unspecified layout policy will default to LAYOUT_LEFT, left-aligned on the Form. If the Item specifies another layout policy, such as LAYOUT_CENTER, then the Form will attempt to accommodate the desired layout for the Item on the Form. As the Items are laid out on the Form, the layout algorithm attempts to keep subsequent Items next to each other, unless any of the following occurs:

- The previous Item has a row break after it.
- The current Item has the LAYOUT_NEWLINE_BEFORE layout hint set.
- The current Item is a StringItem that begins with \n.
- The current Item is a ChoiceGroup, DateField, Gauge, or TextField, and the LAYOUT_2 hint is not set.
- The current Item has a layout flag that differs from the form's current alignment.

A row break occurs after an Item if any of the following occurs:

- The Item is a StringItem that ends with \n.
- The Item has the LAYOUT_NEWLINE_AFTER hint set.
- The Item is a ChoiceGroup, DateField, Gauge, or TextField, and the LAYOUT_2 hint is not set.

If all of this seems confusing, don't panic: the behavior is actually fairly intuitive, and you'll find in practice that a bit of experimentation quickly yields the layout you want.

The Form's other responsibility is to keep a collection of the Items it draws. While the details of that collection are private to the implementation of the Form class, the Form class provides the following methods:

- append: Appends an Item to the Form
- delete: Takes an index and deletes the Item at the specified index in its collection from the Form

- `deleteAll`: Deletes all `Items` in the `Form`
- `get`: Takes an index and returns the `Item` at that index
- `insert`: Takes an index and an `Item` and inserts the `Item` after the specified index
- `set`: Takes an index and an `Item` and replaces the `Item` at the specified index with the new `Item`
- `size`: Returns the number of `Items` in the `Form`

A `Form` can have a listener that its `Items` invoke when they change values. This notification occurs by invoking the `itemStateChanged` method of the listener you register with the `setItemSateListener` method. Note that this *isn't* triggered when you invoke `Command` instances on the form; the `CommandListener` you register using the `setCommandListener` receives `Command` events.

Alerting the User

The `Alert` class provides a screen that shows data to the user and waits for a predetermined amount of time before automatically showing another `Displayable`. Typically, `Alerts` appear full-screen; the application provides a title, optional image, and body text. You can provide your own image, title, body text, and a gauge that replaces the image; Figure 5-9 shows a sample `Alert` with a `Gauge`.

The constructor for an `Alert` can take up to four arguments: a title, text for the alert, an image, and an alert type. The first three are self-explanatory; the fourth is a value from the `AlertType` enumeration indicating whether the `Alert` is an alarm, confirmation, error, warning, or informative alert. Once you create an `Alert`, you should configure it with its time-out; as shown in Listing 5-3, you do this using the `setTimeout` method, specifying the delay in milliseconds.

Figure 5-9. *An Alert with a Gauge*

Listing 5-3. *Configuring the Alert*

```
public Alert get_cannotAddLocationAlert() {
    if (cannotAddLocationAlert == null)
    {
       cannotAddLocationAlert = new Alert("Cannot Add Location");
       cannotAddLocationAlert.setString("An error occurred adding the➥
location you entered. It has not been added.");
       cannotAddLocationAlert.setTimeout(10000);
       cannotAddLocationAlert.addCommand(get_backCommand());
    }
    return cannotAddLocationAlert;
}
```

```
public void add_location( String l ) {
    String locations[];
    int i;
    try {
        locationStore.addLocation( new Location( l, "" ));
        locationList = null;
    } catch (Exception e) {
        getDisplay().setCurrent(get_cannotAddLocationAlert(),
            get_locationList());
    }
    // Refresh the location list lazily.
    }
```

Showing an `Alert` is a little different than showing other `Displayable`s, because you need to specify the `Displayable` that the screen should show after showing the `Alert`. Consequently, you use the `Display.setCurrent` method, which takes both an `Alert` and a subsequent form, like this:

```
Display.getDisplay().setCurrent( alert, nextForm );
```

Interestingly, you can set your own `CommandListener` on an `Alert` using `setCommandListener`, but be careful if you do: it disables the autoadvance feature that takes the screen to the next `Displayable` either when the user dismisses the `Alert` or its timer expires. To restore this functionality, just set a `CommandListener` of `null`.

The Alert `class` has the usual gamut of accessor and mutator methods you'd expect, including the following:

- `setTimeout` *and* `getTimeout`: Set and get the time-out until the `Alert` transitions to the next `Displayable`
- `setImage` *and* `getImage`: Set and get the `Image` instance displayed by the `Alert`
- `setString` *and* `getString`: Set and get the string (not the title!) displayed by the `Alert`
- `setIndicator` *and* `getIndicator`: Set and get the gauge displayed by the `Alert`

■ **Tip** If you set a `Gauge` for an `Alert`, it must be noninteractive, not owned by another `Displayable`, and not have any commands or label, and its layout value must be `LAYOUT_DEFAULT`.

Accepting Copious Amounts of Text

The `TextBox` class provides a full-screen alternative to the `TextField` and shares an interface similar to the `TextField` (only, unfortunately, the MIDP does not break up the text interface into separate interfaces and implementations, as it did for `Choice`, `ChoiceGroup`, and `List`). Figure 5-10 shows a `TextBox`.

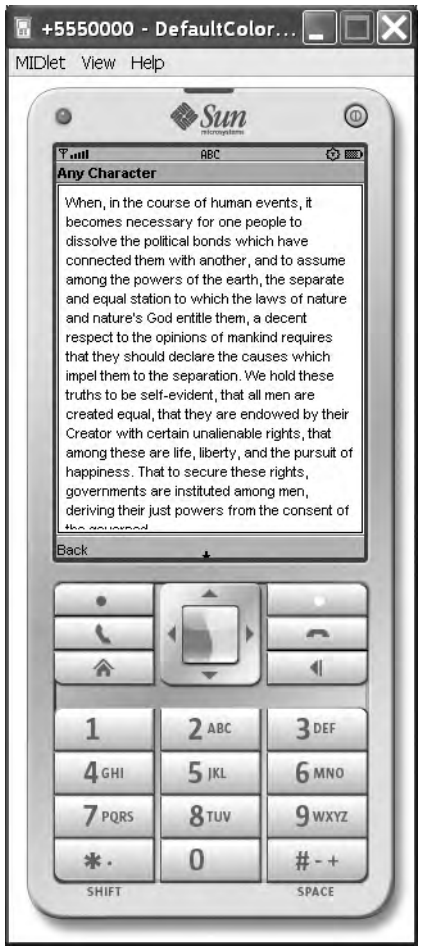

+5550000 - DefaultColor...
MIDlet View Help
Sun
ABC
Any Character
When, in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the laws of nature and nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation. We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable rights, that among these are life, liberty, and the pursuit of happiness. That to secure these rights, governments are instituted among men, deriving their just powers from the consent of
Back
1
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
*.
0
#-+
SHIFT
SPACE

Figure 5-10. *A TextBox*

As you can see from the figure, `TextBox` instances are best for managing the input of large chunks of text, and there's the rub: most MIDP-capable devices don't provide interfaces well suited to text entry. Consequently, I recommend that you avoid using `TextBox` instances if you can. Instead, try to design your application so that it requires as little text input as possible, either by omitting long text entry entirely or by memorizing repetitive text entry and presenting the memorized text as choices in a `List` or `ChoiceGroup`.

Showing Lists of Choices

The `List` class provides you with a way to present full screens of choices, such as full-screen single-selection or multiple-selection lists. Figure 5-11 shows one such `List`, with the corresponding code in Listing 5-4 taken from Sun's Java ME example code.

Figure 5-11. *A List with multiple choices and a single ITEM command*

Listing 5-4. *Code Generating the List Shown in Figure 5-11*

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class ListExample extends MIDlet  {
  public void startApp() {
    final List l = new List("Pizza Toppings", Choice.MULTIPLE);
    l.append("Anchovies", null);
    l.append("Cheese", null);
    l.append("Olives", null);
    l.append("Onions", null);
    l.append("Pepperoni", null);
    l.append("Sausage", null);

    l.addCommand(new Command("Order", "Order Pizza", Command.ITEM, 0));

    l.setCommandListener(new CommandListener() {
          public void commandAction(Command c, Displayable s) {
          boolean isSelected[] = new boolean[ l.size() ];
          int i;
          l.getSelectedFlags( isSelected );
          for ( i = 0; i < l.size(); i++ ) {
            if ( isSelected[ i ] ) {
              System.out.println( l.getString( i ));
            }
          }
        }
      } );

    Display.getDisplay(this).setCurrent(l);
  }
  public void pauseApp() { }
  public void destroyApp(boolean unconditional) { }
}
```

This code also shows the usual way of getting selections from a `Choice` subclass: in the case of multiple lists, simply iterate over a list of `boolean` values obtained from `Choice.getSelectedFlags`. For an exclusive list, it's even easier: just invoke `Choice.getSelectedIndex`, because the user can select only a single item.

■ **Note** Wondering why the `List l` is declared as `final` in Listing 5-4? It's because the variable `l` is referenced in the closure created when you declare the anonymous `CommandListener` subclass. The local class doesn't really access `l`, but instead a private copy of `l`. If `l` were to change after it was declared but before the platform invokes the closure's `commandAction` method, the two notions of `l` would be out of sync with each other.

Working with the Canvas and Custom Items

Despite the flexibility of the `Screen` and `Item` class hierarchy, there are some things you just can't do with these classes. One notable example, of course, is creating a game; moreover, nearly any truly bespoke interface is out of reach, because of the high level of abstraction provided by these classes.

There is another way, however. The MIDP provides the `Canvas`, which is a base class for writing applications that need to handle low-level events and perform low-level drawing. Instances of `Canvas` or its subclass `GameCanvas` encapsulate event and drawing behavior by passing events to the instance and permitting the instance to draw using the `Graphics` object passed to the instance's `paint` method. These classes bring you as close as you can come to the bare hardware of a MIDP platform.

Sometimes, though, you may want to take advantage of all the abstractions that the `Screen` and `Item` classes provide, yet you may need a custom `Item` to present particular data. You can do this using a `CustomItem` subclass, which lets you implement a UI widget that works within the framework established by the `Screen` and `Item` subclasses.

■ **Caution** Don't venture into Java ME programming thinking the `Canvas` class is like the `Canvas` class in Java SE! They are two very different beasts.

Controlling Drawing Behavior with a Custom Canvas

Subclassing the `Canvas` class gives you the opportunity to manage events and drawing behavior at the lowest level. An implementation of `Canvas` must be able to do the following:

- *Handle events*: The `Canvas` class receives low-level events, including key-press, repeat, and release events, as well as pointer-press, drag, and release events.

- *Handle commands*: As with other `Displayable` subclasses, your `Canvas` implementation inherits the methods pertaining to the `Command` infrastructure, including `addCommand`, `removeCommand`, and `setCommandListener`.

- *Draw*: Subclasses of `Canvas` must implement the `paint` method, which takes a `Graphics` instance that you use to draw on the `Canvas`.

The downside to using the `Canvas` class is that event handling is not as portable as the `Command` and `CommandListener` classes are. For every keystroke, your `Canvas` receives a `keyPressed` invocation and a `keyReleased` invocation, and possibly one or more `keyRepeated` invocations between the two. The system passes these methods a *key code*—that is, an integer indicating the key that you pressed. The `Canvas` class defines constant members for keys common to all MIDP devices, including the number keys, the arrow keys, four game keys, and the fire key (selection key). There may be overlap between these keys; for example, the game keys may actually be number keys.

Unfortunately, other keys that aren't common to all MIDP devices may have different key codes, so if you're designing a more complex `Canvas` subclass that uses many keys, or one that requires alphanumeric input, you'll end up needing to write code for each device on which your application will run. One way to do this is to abstract key-code handling from event handling, in much the same way that the MIDP platform does; instead of examining key codes directly, use a function to map the key codes and the logical actions that the key codes represent, so that you can map multiple key codes to a single logical event.

Your `Canvas` subclass can receive pointer events, too, provided that your application is running on a platform that supports some kind of pointer (mouse or stylus, presumably). Some devices have no pointer; you can determine the support for pointer events by invoking `hasPointerEvents`, which returns `true` if the device supports pointer-press and release events. In a similar vein, the `hasPointerMotionEvents` method of `Canvas` returns `true` if the device supports drag events from the pointer. Assuming there's support, you receive events by overriding the `pointerPressed`, `pointerDragged`, and `pointerReleased` methods; each of these receives two integers, the *x* and *y* coordinate for the event.

At any time, the system can signal that the `Canvas` should redraw itself by invoking its `repaint` method. This method comes in two varieties: one that takes the bounds of the rectangle to redraw, and one that takes no arguments but is equivalent to invoking `repaint` with the entire `Canvas` bounds. The `repaint` method doesn't do any drawing, however; that's the responsibility of the `paint` method. The `repaint-paint` flow is asynchronous; `repaint` indicates to the object that it's dirty and should plan on repainting, while the actual drawing by `paint` occurs at an unspecified (hopefully short!) time in the future. This permits applications and the system from triggering needless redraw operations, and it keeps clients from blocking on the `Canvas`'s `paint` operation.

You override the `paint` method to actually perform drawing. The caller passes this method a single argument: an instance of `Graphics` that you use to actually perform drawing. Your `paint` operation should repaint every pixel within the region defined by the `Graphics` object's clipping, because it may have resulted from multiple `repaint` calls. Note that the `Graphics` instance is only valid within your `paint` method; you definitely should *not* cache aside the `Graphics` instance you receive to use after exiting the `paint` method.

GRAPHICS OPERATIONS AND DOUBLE BUFFERING

On early Java ME devices, graphics operations to the screen were slow, and applications that did a lot of painting showed visible artifacts such as flickering or tearing of the image as the platform interleaved drawing operations with screen refreshes. Although a much rarer problem today, this can still be a challenge, especially if you're implementing a fast-paced game or similarly demanding application.

To avoid visual artifacts from display updates while drawing, you use a technique called *double buffering* or *ping-pong buffering*, in which you perform all of your drawing operations on an offscreen bitmap, and then when you're done with all of the drawing operations, you transfer that image to the display. Newer versions of the MIDP support double buffering; you can query the `Canvas` directly by invoking `Canvas.isDoubleBuffered`. If this method returns `true`, the MIDP implementation will render the results of your `paint` operation into an offscreen bitmap and transfer the bitmap at appropriate times to the display's framebuffer, preventing visual artifacts.

If the platform does not support double buffering, you can implement double buffering yourself. Instead of drawing with the `Graphics` object that the platform passes to your item's `paint` method, create an instance of `javax.microedition.lcdui.Image` to buffer all of your drawing operations. Then, invoke the new image's `getGraphics` method to obtain the `Graphics` object associated with the image, and do all the drawing with that `Graphics` object. When you're done, render the image directly to the screen using the `Graphics` object passed to your `paint` method, like this:

```
void paint(Graphics g) {
    int w = getWidth();
    int h = getHeight();
    Image buffer = Image.createImage(w, h);
    Graphics bg = buffer.getGraphics();
    bg.drawRect(0, 0, 10, 10);
    g.drawImage( buffer, 0, 0 );
}
```

If you're only doing simple graphics updates or updating a small region of the screen, the memory overhead imposed by double buffering may be more expensive than it's worth and cause performance penalties of its own. As a result, you should test your code carefully to determine if double buffering is actually necessary.

The Graphics class contains the usual gamut of interfaces for drawing to the Canvas that you might expect, including methods to draw strings, lines, arcs, individual pixels, and images. Its interface is similar to, but not the same as, the Java SE Graphics class, so it's best to check the MIDP documentation before writing code that uses this class.

Creating a Custom Item for a Screen

The CustomItem class gives you an abstract class from which to implement a new interactive Item that you can place on a Form. CustomItem instances must be able to do the following:

- Determine their appropriate size
- Draw the contents of the item
- Respond to events generated by keys, pointers, and traversal of its internal focusable subitems (entry and exit of each focusable subitem)
- Invoke the notifyStateChanged method when the value has changed

Item objects, including CustomItem subclasses, interact with their parent object via the notion of a minimum and preferred size. The former size is the smallest size the parent may give the CustomItem, while the latter is the size the CustomItem would like to occupy on the parent. For a given CustomItem, the content size describes the actual region the parent has allocated for its drawing; a CustomItem must draw its contents in that area. To make matters simpler, the parent communicates the content region in coordinates relative to the CustomItem—that is, the upper-left corner of the CustomItem is (0,0). Sizing from your CustomItem to the parent is passed via the following methods:

- getMinContentHeight *and* getMinContentWidth: Let you specify your CustomItem's minimum height and width, respectively
- getPrefContentHeight *and* getPrefContentWidth: Let you specify the preferred height and width for your CustomItem (possibly based on its current contents)

The parent form passes your CustomItem's *actual* bounds—the content bounds—to your paint method. Like a Canvas, you override paint to provide the code that repaints the CustomItem's region. You must paint every pixel clipped by the provided Graphics instance. Unlike the Canvas, whose bounds are set by methods, your content bounds are passed as arguments to the paint method. You can also schedule redraws using the repaint method, just as you might a Canvas.

The events a CustomItem may receive depend on the MIDP implementation, which is a barrier to application portability. Your implementation may determine which events it supports by invoking CustomItem.getInteractionModes; the resulting integer is a bit mask

that specifies which events the device will pass to your `CustomItem`. The actual values for the bit mask are provided as fields of `CustomItem`, and include the values `KEY_PRESS`, `KEY_RELEASE`, `KEY_REPEAT`, `POINTER_PRESS`, `POINTER_DRAG`, and `POINTER_REPEAT`. For each kind of event you intend to support, you must implement the appropriate method, which the system will invoke to inform your subclass that there's an event of that type ready for it to process. These methods are the same as for the `Canvas` class:

- `keyPressed`: Handles key presses by key code
- `keyReleased`: Handles key releases by key code
- `keyRepeated`: Handles repeated key presses (when you press and hold a key) by key code
- `pointerPressed`: Handles a pen-down or mouse-down at a coordinate
- `pointerDragged`: Handles a user dragging the pointer to a new coordinate
- `pointerReleased`: Handles a pen-up or mouse-up at a coordinate

In addition to these events, you must also handle focus events, which the MIDP specification confusingly calls *traversal* operations (presumably because they're generated as you traverse the parent). Traversal operations give you a way of knowing when your `CustomItem` is active and from which direction you arrived at the `CustomItem`. For example, a `CustomItem` that displays richly formatted text that can scroll outside its content area might wish to show the last pieces of text if it's entered from below, or the first piece of text it contains when entered from above. To do this, you must implement `CustomItem.traverse`, which the caller gives four pieces of information.

The first argument indicates the direction from which you navigated into the `CustomItem`. The second and third arguments give the width and height of the viewable area that the item's container has given its item. From this, you can assume that this is the largest bound your item will be given to draw, although its actual content bounds will be passed to the `paint` method. The final argument is an array indicating a rectangle bound in the form `[x, y, w, h]`. When the container calls `traverse`, it contains the rectangle currently visible; when your `traverse` method exits, it should contain the bounds of the rectangle relevant to the viewer. Thus, if your `CustomItem` contains more information than its content area, you should pass the region to be displayed in this rectangle.

The view system may invoke `traverse` to indicate that a `CustomItem` has focus, or, once focused, to indicate that a subitem in the `CustomItem` should be focused. For example, a `CustomItem` implementing a grid of cells (such as a spreadsheet) receives `traverse` invocations for each directional arrow press, and it draws its contents so that the currently selected cell is indicated in some way. To keep your `CustomItem` focused and receiving traversal events, your `CustomItem.traverse` method must return `true`. To indicate that navigation of your `CustomItem` is complete—that the user has traversed out of the item—return `false` from `traverse`.

Finally, if your `CustomItem` has the concept of state—that is, if it contains a value—and the value changes, you must call `notifyStateChanged` to let any listener on the `CustomItem` know that its state has changed.

Implementing a Custom Item

Let's wrap this up with a concrete example. Say the weather application obtains specific weather conditions—sunny, cloudy, rainy, snowy, and so forth—from a remote server, and you want to present that information in a graphic format on the `Form` that shows the current weather. You could do this one of two ways. You could simply select a particular image based on the conditions report, or you could encapsulate that functionality in a `CustomItem` responsible for drawing the image associated with particular weather conditions. In practice, which you choose may not make *that* much difference, but for the purposes of this discussion, let's assume the encapsulation in a `CustomItem` is better, because it provides clear encapsulation and responsibility for the data presentation in a single class. Listing 5-5 shows some of the code necessary to implement the `WeatherItem`, a class that implements the functionality inherited from `CustomItem`.

Listing 5-5. *The WeatherItem, a CustomItem Subclass for Displaying Weather Conditions*

```
package example.wxitem;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;


public class WeatherItem
       extends CustomItem {
    private String title;
    private Display display;
    private int width, height;
    private boolean hasFocus = false;
    private int conditions;

    // States (not mutatable by caller!)
    public final int    SUNNY = 1;
    public final int    PARTLY_CLOUDY = 2;
    public final int    CLOUDY = 3;
    public final int    SHOWERS = 4;
    public final int    RAIN = 5;
    public final int    FLURRIES = 6;
    public final int    SNOW = 7;
    public final int    SLEET = 8;
```

```
// Default size
final private int DEFAULT_HEIGHT = 64;
final private int DEFAULT_WIDTH = 64;

// Colors
final private int WHITE = 0xFFFFFF;
final private int YELLOW = 0xFFFF00;

public WeatherItem( String t, Display d ) {
    super(t);
    title = t;
    display = d;
    hasFocus = false;
    width = DEFAULT_HEIGHT;
    height = DEFAULT_WIDTH;
    conditions = 0;
}

public void setConditions( int c ) {
    conditions = c;
}

public int getConditions( ) {
    return conditions;
}

protected int getMinContentHeight() {
    return height;
}

protected int getMinContentWidth() {
    return width;
}

protected int getPrefContentHeight(int w) {
    if ( w < 0 )
        return height;
    else
        return height * w / width;
}
```

```
protected int getPrefContentWidth(int h) {
    if ( h < 0 )
        return width;
    else
        return width * h / height;
}

protected void paint(Graphics g, int w, int h) {
    // Always paint SOMETHING
    g.setColor(WHITE);
    g.fillRect(0, 0, w, h);
    switch( conditions )
    {
        case SUNNY:
            drawSun(g, w, h);
            break;
        case PARTLY_CLOUDY:
            drawSun(g, w, h);
            // FALL-THRU
        case CLOUDY:
            drawCloud(g, w, h);
            break;
        case SHOWERS:
            drawSun(g, w, h);
            // FALL-THRU
        case RAIN:
            drawCloud(g, w, h);
            drawRain(g, w, h);
            break;
        case FLURRIES:
            drawSun(g, w, h);
            // FALL-THRU
        case SNOW:
            drawCloud(g, w, h);
            drawSnow(g, w, h);
            break;
        case SLEET:
            drawCloud(g, w, h);
            drawRain(g, w, h);
            drawSnow(g, w, h);
            break;
```

```
            default:
                drawUnknown(g, w, h);
                break;
        }
    }

    private void drawSun(Graphics g, int w, int h) {
        int x, y, min, r;
        min = Math.min(w, h);
        r = 3 * min / 4;
        x = ( w - r ) / 4;
        y = ( h - r ) / 4;

        g.setColor( YELLOW );
        g.fillArc(x, y, x + r, y + r, 0, 360);
    }

    private void drawCloud(Graphics g, int w, int h) {
        // Drawing code here.
    }

    private void drawRain(Graphics g, int w, int h) {
        // Drawing code here.
    }

    private void drawSnow(Graphics g, int w, int h) {
        // Drawing code here.
    }

    private void drawUnknown(Graphics g, int w, int h) {
        // Drawing code here.
    }

    protected boolean traverse(int dir, int viewportWidth, int viewportHeight,
                                            int[] visRect) {
        hasFocus = !hasFocus;
        if ( hasFocus )
        {
            visRect[ 0 ] = 0;
            visRect[ 1 ] = 0;
            visRect[ 2 ] = width;
            visRect[ 3 ] = height;
        }
```

```
        return hasFocus;
    }
}
```

The class begins with the constructor, which picks a (somewhat arbitrary) default width and height for the `WeatherItem` and initializes its `conditions` field—which stores what weather indicator should be drawn—to `0`. `WeatherItem` exports the `conditions` field as a property through the `setConditions` and `getConditions` methods, which admittedly could use some bounds checking.

The item returns the default bounds set by the constructor as the minimum bounds, and uses the aspect ratio set by those bounds to compute the preferred content height and width. Note that the parent of the item may invoke the `getPrefContentHeight` and `getPrefContentWidth` methods with a value of `-1` when setting up their layout, and to that the `CustomItem` should respond with a default desired size.

The `paint` method does just that, switching on the weather indication and drawing a sun, clouds, rain, or snowflakes as appropriate. The implementation assumes that each of these images are layered, either as bitmaps or drawn using the `Graphics` context and 2D vector graphics. For example, `drawSun` draws a circle three-quarters the size of the `WeatherItem` centered within the `WeatherItem`. Other drawings might use the `Graphics` methods or `Graphics.drawImage` to draw from PNG files stored within the MIDlet's JAR file.

The implementation of `traverse` is trivial, because it only needs to track whether the item has been traversed into (focused) or out of (unfocused). Clearly, when the `WeatherItem` is focused, the rectangle of the item to draw should be its entire rectangle; when it's defocused, the `WeatherItem` doesn't return a rectangle.

Wrapping Up

Although in no way compatible with Java SE, the MIDP provides a versatile collection of high-level user-interface items and a more flexible low-level alternative. You can create flexible, easily ported user interfaces using the `Screen` subclasses `Alert`, `TextBox`, `List`, and `Form`, along with the various visible `Item` classes including `TextBox`, `ChoiceGroup`, `StringItem`, `ImageItem`, and `Gauge`. Moreover, if you need to, you can extend the hierarchy of items by creating custom items by subclassing `CustomItem`, providing your own event handling and drawing behavior for a new item. For lower-level access to the event and drawing system, you can subclass the `Canvas` class, which lets you receive pointer and keystroke events directly, as well as draw on the screen using instances of the `Graphics` class. Because both the `Canvas` and `Screen` classes implement the same parent, `Displayable`, you can mix and match `Canvas`-based and `Screen`-based displays in the same application.

CHAPTER 6

Storing Data Using the Record Store

At some point, nearly every MIDP application will need to store data. The original version of the MIDP did not offer access to the file system (which I discuss in the next chapter), but it did introduce a complementary concept: the *record store*. Unlike a file, which can store anything, a record store stores groups of similarly structured items, although in truth it falls short of meeting all the purposes of a database.

In this chapter, you'll learn about the record store as it's provided through the `javax.microedition.rms` interface. After reading this chapter, you will understand how the record store represents records; how to create, remove, and share record stores; and how to access, mutate, and remove records. Along the way, you will see one application of the record store as I show you how to use it to store data within the WeatherWidget application.

Peeking Inside the Record Store

Through the `javax.microedition.rms` interface, MIDlets have access to multiple uniquely named record stores. Each *store* contains a collection of *records*. MIDlet suites may mark a record store as *private*, meaning that only MIDlets in the MIDlet suite that creates the store can access the store, or *shared*, meaning that any MIDlet on the device can access the store. The MIDP implementation keeps record stores in a nonvolatile region of the device, such as a hard drive, flash file system, or RAM backed up by battery, so that data persists between MIDlet instances. When a MIDlet is removed, all record stores that the MIDlet created are removed as well. Figure 6-1 shows a schematic relationship between MIDlet suites, MIDlets, and record stores.

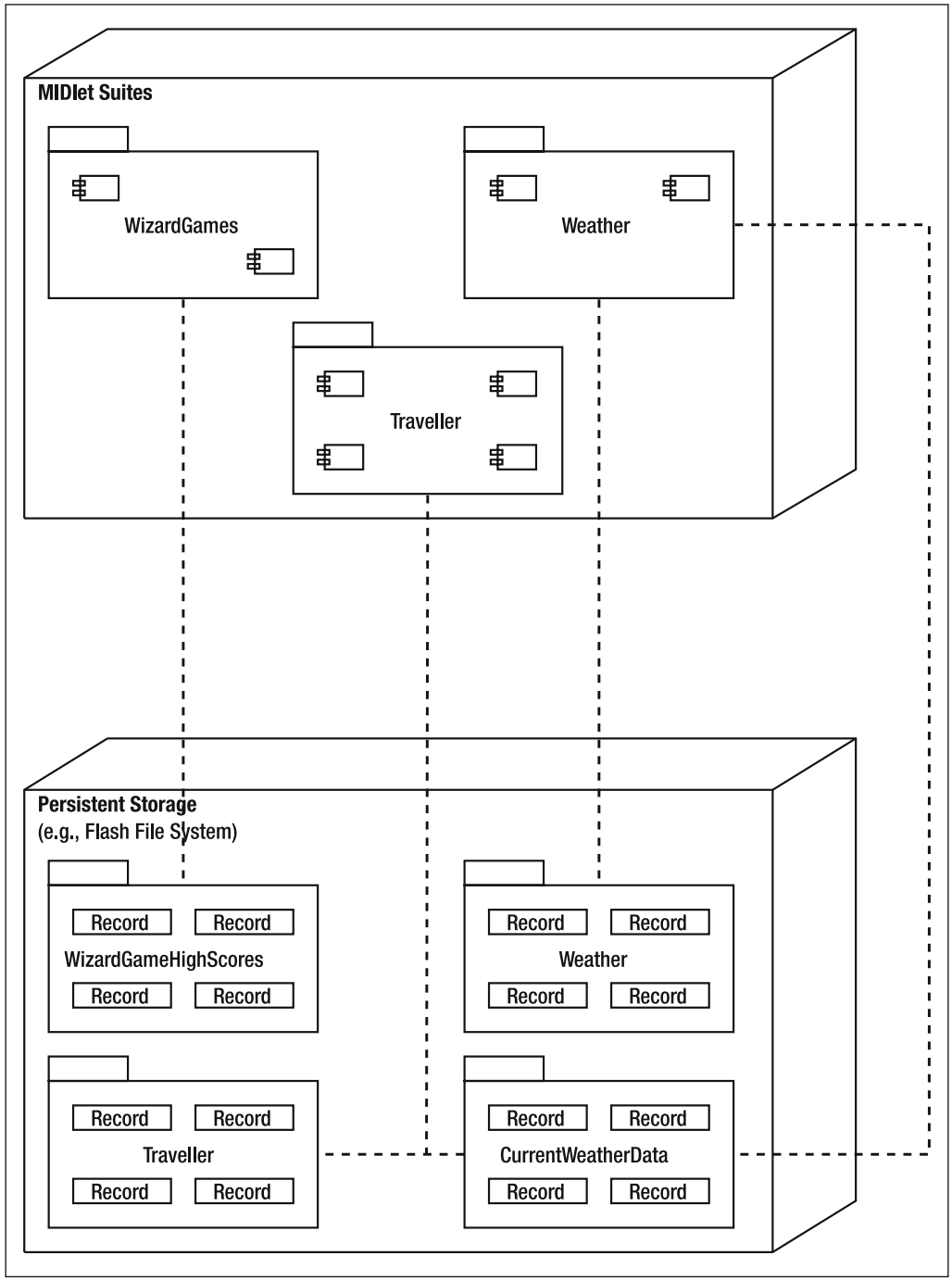


Figure 6-1. *The relationship between MIDlet suites, MIDlets, and record stores*

As shown in Figure 6-1, the MIDlet suites in the example application follow these rules:

- The `WizardGames` MIDlet suite uses only the `WizardGameHighScores` record store, which is private to `WizardGames`.
- The `Weather` and `Traveller` MIDlet suites each have a private record store for configuration and other data.
- The `Traveller` and `Weather` MIDlet suites read the current weather forecasts from the `CurrentWeatherData` public record store.

Of course, both the `Traveller` and `Weather` MIDlet suites must understand the representation schema for records in the `CurrentWeatherData` public record store. You can make this happen most easily by sharing a class, because code can't be shared between MIDlets.

Record store names must be unique. To help guarantee this, the Java ME runtime uses a MIDlet suite's name as part of the name for a record store, so that you only need to guarantee uniqueness when naming multiple stores within your application. Note that the names are case sensitive and represented as Unicode strings, so you can localize them as you would any other application string (although they typically don't appear anywhere in the user interface unless your MIDlet shows a list of its record stores).

Records within the record store are given a unique index, which is the record's primary key. This index starts with `1` and increments so that after creating record `n`, the next record you create is given the index `n+1`. Removing a record does not recycle the record ID; once a record ID is used, it's permanently assigned, even if the value to which it's assigned is `null`.

Implementations of the record store must guarantee atomicity across a single thread, but do not guarantee atomicity across multiple threads. Consequently, if your MIDlet accesses the record store from multiple threads, be sure to synchronize access to records in the store.

Using the Record Store

There's actually very little you can do with a record store, because most of the work you perform is with records within a store. Consequently, you can open or close a record store, as well as remove a record store. Finally, you can check the version of a record store, so you can ensure that your application uses the right record representation when reading records from and writing records to the store.

All of the methods in this section belong to the `javax.microedition.rms.RecordStore` class.

Opening and Closing a Record Store

Given the name of a record store, opening one is easy: simply invoke the `RecordStore.openRecordStore` method, passing the name of the store and stating whether the store should be created if it doesn't exist. This method returns an instance of `RecordStore` you use to access records in the open store. For example,

```
RecordStore.openRecordStore("Weather", true);
```

creates the record store named `Weather` for the current MIDlet and returns an instance of `RecordStore`. This method can throw one of the following exceptions, so you best be prepared to handle it:

- `RecordStoreException`: The method throws this exception in response to a general error creating or opening the record store.
- `RecordStoreNotFoundException`: The method throws this exception if it can't find the record store you named and the creation flag is `false`.
- `RecordStoreFullException`: The method throws this exception if the record store is full.
- `IllegalArgumentException`: The method throws this exception if the name is invalid.

Other `RecordStore.openRecordStore` methods take additional arguments that you must use if you want to share your record store with multiple applications. In addition to specifying the name of the record store and whether or not it should be created, you pass an access mode (`AUTHMODE_PRIVATE` or `AUTHMODE_ANY`, meaning that any MIDlet can access the record store) and a final argument indicating whether the store should be available for read-only access (`false`) or read and write access (`true`) to other MIDlets. This interface lets you create a shared record store; the final overloaded `RecordStore.openRecordStore` method lets you open a shared record store from a different MIDlet. It takes three arguments: the name of the store, the name of the MIDlet suite vendor exporting the record store, and the name of the MIDlet suite exporting the record store.

As a result, for two MIDlets to share record stores, *all* of the following must occur:

- The MIDlet creating the record store to share must create it using `RecordStore.openRecordStore`, passing `true` to create the record store, `AUTHMODE_ANY` to permit other MIDlets to access the record store, and `true` if the shared record store is to be writable by other applications.
- The author of the MIDlet creating the record store to share must publicize the format of records in the record store (ideally through a class that client applications can use to access the store directly) as well as the vendor name and MIDlet suite name used by the MIDlet creating the record store.

- Other MIDlets outside the MIDlet suite of the MIDlet creating the record store must open the store using `RecordStore.openRecordStore`, passing the name of the record store, the name of the creating MIDlet suite vendor, and the name of the MIDlet suite that originally created the store.

Once you're done with a record store, you must close it, signaling to the underlying MIDP implementation that you're done using the resources consumed by the record store. You do this by invoking `closeRecordStore` on the instance provided when you opened the record store.

Removing a Record Store

Although the MIDP removes the record stores associated with a MIDlet suite when the suite is deleted, sometimes you want to delete a record store yourself. You do this by invoking the `RecordStore.deleteRecordStore` method, which passes the name of the record store to delete. The record store must be owned by the MIDlet suite of the MIDlet making the request, and it may throw an exception if the store is open or any other error occurs. It can throw the following exceptions:

- `RecordStoreException`: In response to a general error when removing the record store, such as if the record store is still open
- `RecordStoreNotFoundException`: If the record store you named couldn't be found

As you'll learn in the upcoming section "Accessing Records in the Record Store," you can listen for changes to the record store, causing the record store to invoke a method when changes occur. You should be aware, however, that the record store does not invoke the method you provide when you delete the record store; in deleting the record store, the store and its records are irretrievably deleted, and the system invokes no listeners.

Obtaining Information About a Record Store

While you're not privy to the underlying implementation details of the record store, there *are* some details about a specific record store you can access. Of course, you typically know the *name* of the record store with which you're working. An application that takes a document-oriented approach to its data—say, one that tracks expense reports—might choose to represent each document as a record or as a record store. If the latter, then users may be responsible for naming record stores, such as "Sales Trip London" or "Sun JavaOne Trip." Of course, once you open the gates to user-named record stores, you need a way to determine *what* record stores exist; you do this using the `RecordStore.listRecordStores` method, which returns an array of `String` objects, each containing the name of a specific record store belonging to the MIDlet suite. If the suite's MIDlets have not created any applications, this method will return `null`.

For a given record store, you can determine the last time the store was modified by using its `getLastModified` method, which returns the last modified time in milliseconds since the start of the system epoch (e.g., 1 January 1970). You can turn around and use this time with the utilities in `java.util.Date` and `java.util.Calendar` to manipulate the result, as well as `javax.microedition.lcdui.DateField` to display the result. Other details you can obtain include the size (in bytes) of the store, which is returned by its `getSize` method, and the number of records, which is obtained using the `getNumRecords` method. All of these can throw a `RecordStoreNotOpen` exception, indicating that you must open the store prior to obtaining this information.

Finally, record stores support the notion of versioning, but it's not the sort of versioning you might expect. A record store's version doesn't indicate some canonical version you apply when you change the format of its records or the application accessing the record store; instead, it's a monotonically increasing number that indicates the number of changes (via the `addRecord`, `setRecord`, and `deleteRecord` methods). The initial version of a store is an implementation-specific value, so while you can't use a store's version to count the number of changes to the store, you can use it to peek at a store and see if it has changed since the last time you accessed its contents. This can come in handy when sharing a record store between applications.

Accessing Records in the Record Store

As you saw in Figure 6-1, the organization of records in the record store is free form. Unlike a database, where you can define primary keys and search on specific fields, the closest you get to a primary key in a record store is a record's index. The first record you create within a specific record store always has a record ID equal to 1; subsequent records are assigned a record ID one greater than the record added previously. As I noted in the previous section, removing a record does not recycle its record ID. For example, consider the following sequence of events:

1. A MIDlet creates a new record store, the `SampleRecordStore`.
2. The MIDlet adds a record to the record store. Its ID is 1.
3. The MIDlet adds two more records. These records have IDs of 2 and 3, respectively.
4. The MIDlet now deletes record 2.
5. The MIDlet adds another record. This record has an ID with the value 4.

After this sequence of events, the record store will have three records, with IDs 1, 3, and 4, respectively.

A record itself is just a bag of bytes. The good news is that you're free to implement any structure you see fit on a record; the bad news is that it's up to you to define a structure that supports your application and scales well. The MIDP implementation lacks the serialization support found in `java.io.Serializable`, meaning you need to implement your own serialization. (The interface defined by `java.io.Serializable` doesn't really fit the notion of records in the record store, either, because there are no lower-level filter streams to help with serialization, and the object of record serialization is to serialize to a bag of bytes rather than an output stream.)

This means that you need to be familiar with Java streams, particularly the `java.io.ByteArrayInputStream`, `java.io.ByteArrayOutputStream`, `java.io.DataInputStream`, and `java.io.DataOutputStream` classes. The process isn't as daunting as it first appears.

To preserve a record, you first create a `DataOutputStream` and a `ByteArrayOutputStream`, as shown in Listing 6-1.

Listing 6-1. *Creating a DataOutputStream and a ByteArrayOutputStream*

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
```

Next, you write the record's fields to the `DataOutputStream` using one or more of the following methods:

- `write`: Takes an array of bytes, an offset, and a number of bytes to write, and writes those bytes to the stream
- `writeBoolean`: Writes a `boolean` to the stream
- `writeByte`: Writes a single byte to the stream
- `writeChar`: Writes a character to the stream as a two-byte value with the high byte first
- `writeChars`: Writes a `String` as a sequence of characters
- `writeDouble`: Converts the `double` to a `long` using `doubleToLongBits` and writes the `long` to the stream as an eight-byte value
- `writeFloat`: Converts the `float` to an `int` using `floatToIntBits` and writes the `int` to the stream as a four-byte value
- `writeInt`: Writes an integer with the high byte first
- `writeLong`: Writes a `long` integer with the high byte first
- `writeShort`: Writes a `short` integer with the high byte first
- `writeUTF`: Writes a `String` using Java-modified UTF-8 encoding (in a machine-specific way)

Once you write the record fields, you can obtain an array of bytes from the `ByteArrayOutputStream` by invoking its `toByteArray` instance method, as shown in Listing 6-2.

Listing 6-2. *Invoking the toByteArray Instance Method*

```
dos.writeUTF(record);
byte[] bytes = baos.toByteArray();
```

With the byte array in hand, you can add it to the store using the `addStore` method, as I show you in the "Adding a Record" section later in this chapter. Of course, it's a good idea to nil out all three (the byte array `bytes`, the `ByteArrayOutputStream` instance, and the `DataOutputStream` instance) once you're done with them, to tell the garbage collector that the memory they consume may be released. This can be especially helpful on older devices with a small heap.

Not surprisingly, deserialization is the opposite of serialization. Begin by creating instances of `ByteArrayInputStream` and `DataInputStream` from the bytes in a record, as shown in Listing 6-3.

Listing 6-3. *Creating ByteArrayInputStream and DataInputStream*

```
ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
DataInputStream dis = new DataInputStream(bais);
```

Next, you can use the following methods from the `DataInputStream` one at a time to read the data you wrote back from the record:

- `read`: Reads an array of bytes from the stream
- `readBoolean`: Reads a `boolean` from the stream
- `readByte`: Reads a single byte from the stream
- `readChar`: Reads a character from the stream as a two-byte value with the high byte first
- `readChars`: Reads a `String` as a sequence of characters
- `readDouble`: Reads a `long` from the stream as an eight-byte value and converts the `long` to a `double`
- `readFloat`: Reads an `int` from the stream as a four-byte value and converts the `int` to a `float`

- `readInt`: Reads an integer with the high byte first
- `readLong`: Reads a long integer with the high byte first
- `readShort`: Reads a short integer with the high byte first
- `readUTF`: Reads a string using Java-modified UTF-8 encoding (in a machine-specific way)

Of course, you must read the fields in the same order as they were originally written.

Finally, you can process the fields into a new object using the object's constructor, or whatever other data structure you choose to use to represent your data while in memory. As with writing, it's a good idea to nil out references to the byte array `bytes`, the `ByteArrayInputStream` instance, and the `DataInputStream` instance.

Of course, you can take a more sophisticated approach if your application requires. One common pattern is to tag each field you write with a *field type* that identifies the field, so that your MIDlet can read fields out of order. I show you how to do that in the last section of this chapter.

Adding a Record

Once you have an array of bytes in hand that you want to store as a record, creating a new record is as easy as invoking the open record store's `addRecord` method. This method takes three arguments:

- The array of bytes containing the data that consists of the record
- The offset to the first byte of the record in the array of data
- The number of bytes in the array that makes up the record

This approach—passing not just a bag of bytes but an offset into the same bag, and a count of the number of bytes that make up the record—lets you construct multiple records in one buffer if need be, and can help reduce thrash on the platform's memory manager.

The `addRecord` method can throw one of the following exceptions:

- `RecordStoreNotOpenException`: Indicates that the record store you want to add a record to isn't open
- `RecordStoreFullException`: Indicates that the record store or the underlying medium has no more room in which to complete the operation.
- `RecordStoreException`: Indicates a general failure related to the open record store
- `SecurityException`: Indicates that the MIDlet is not authorized to write to the record store

For the obsessively curious—or if you're tracking record IDs for some reason, such as to link records in two different record stores—you can get the next record ID to be assigned using the open store's `getNextRecordID` method. This can throw one of the following exceptions:

- `RecordStoreNotOpenException`: Indicates that the record store you want to add a record to isn't open
- `RecordStoreException`: Indicates a general failure related to the open record store

Of course, this isn't atomic: there's no guarantee that if you invoke this, do a bit of stuff, and then insert a record that the ID of the inserted record actually *will have* this ID, because the platform is multithreaded, and another thread within your application may be using the store! So it's up to you to manage thread contention across IDs if you use this interface.

Retrieving a Record

Fundamentally, there are two ways to retrieve a record: by its ID, or by enumerating across all records in the store. If you know a record's ID, you can fetch it directly using the open record store's `getRecord` method and passing the record ID. This method returns the record as an array of bytes. Of course, this method may fail; when it does, it throws one of the following exceptions:

- `RecordStoreNotOpenException`: Indicates that the record store you want to add a record to isn't open
- `InvalidRecordIDException`: Indicates that the record ID is invalid
- `RecordStoreException`: Indicates a general failure related to the open record store

Once you have the bytes that make up your record, it's up to you to deserialize your record from those bytes; as I note in the opening to this section, you'll likely use the `ByteArrayInputStream` and `DataInputStream` classes to do this.

Enumerating a Record

Often, you don't know the ID of a record when you want to fetch it. This may be because you're showing a list of records (such as a list of locations for the weather application, or a list of expenses in an expense report) or because your record structure doesn't require any sort of coherent index. In this case, you need an enumerator, and the MIDP record store has you covered with the record store's `enumerateRecords` method. The `enumerateRecords` method takes three arguments:

- `RecordFilter`: An instance is responsible for selecting which records should be included in the enumeration.
- `RecordComparator`: An instance is responsible for comparing two records; the record store enumerator uses this to determine the sort order of the returned records.
- `boolean`: When `true`, `boolean` indicates that the enumerator should remain current during enumeration to any changes in the record store.

The method returns an instance of `RecordEnumeration`, upon which you can invoke `nextRecord` repeatedly to obtain subsequent records sorted in the order dictated by the `RecordComparator` you provided (if any). You can also invoke `previousRecord` to obtain the record previously returned by `nextRecord` (or the last record of the enumeration if you have not obtained any records using `nextRecord`).

The `RecordFilter` interface your record filter must match has a single method, `match`, which takes an array of bytes (a record from the record store in its serialized form) and returns `true` if the record matches the criteria you define. The `RecordComparator` interface defines the `compare` method, which takes two arrays of bytes (each a record from the record store in its serialized form) and returns one of `RecordComparator.EQUIVALENT`, `RecordComparator.FOLLOWS`, or `RecordComparator.PRECEDES`, depending on whether the first record is equivalent to, comes after, or comes before the second interface. Listing 6-4 shows a purely fictitious example.

Listing 6-4. *Filtering Record Store Entries with an Enumeration*

```
RecordStore rs = RecordStore.openRecordStore( store, true );
RecordEnumeration re = rs.enumerateRecords(
    new RecordFilter() {
        public boolean match(byte[] r) {
            return true;
        }
    },
    new RecordComparator() {
        public int compare( byte[] r1. byte[] r2) {
            return RecordComparator.EQUIVALENT;
    }, false);
```

As intuition would dictate, this *isn't* the best way to traverse all records if order is unimportant. Instead, the most efficient use of `enumerateRecords` is to invoke it without a filter or comparator (just pass `null`). This returns an enumeration that returns each element of the store in an undefined sequence. If you need to visit all records of the store in a specific order, specify a comparison function but no filter; if you'd like to filter out some criteria but order is unimportant, specify a record filter but no record comparator.

The `enumerateRecords` method can throw a `RecordStoreNotOpenException`, if you've closed the record store before invoking it.

Updating a Record

To update a record in the record store, you must have its ID. With its ID in hand, you can use the record store's `setRecord` method to set new contents for the record, just as if you were adding a record. The method takes four arguments:

- The ID of the record to modify
- An array of bytes from which to draw the record
- An offset into the array indicating the first byte of the record
- The number of bytes (counting from the indicated offset) that comprise the record

As with other record manipulations, this method can throw an exception, including any of the following:

- `RecordStoreNotOpenException`: Indicates that the record store you want to add a record to isn't open
- `InvalidRecordIDException`: Indicates that the record ID is invalid
- `RecordStoreFullException`: Indicates that the operation cannot be completed because the record store or its underlying medium is full
- `RecordStoreException`: Indicates a general failure related to the open record store
- `SecurityException`: Indicates that the MIDlet is not authorized to write to the record store.

Removing a Record

To remove a record from the record store, simply call `deleteRecord`, passing the record ID of the record to be deleted. This method throws one of the following exceptions if an error occurs:

- `RecordStoreNotOpenException`: Indicates that the record store you want to add a record to isn't open
- `InvalidRecordIDException`: Indicates that the record ID is invalid

- `SecurityException`: Indicates that the operation cannot be completed because the MIDlet has read-only access to the record store
- `RecordStoreException`: Indicates a general failure related to the open record store

The word *delete* in the interface name is a bit of a misnomer. You have no control over how the record store manages removal; record store compaction may occur on a removal operation, or it may not. Moreover, the interface does *not* recycle the ID assigned to the deleted record; once a record is deleted, the ID remains used. Querying for that ID returns a `null` record or an exception indicating an invalid record ID.

Counting Records

If you're getting ready to preprocess a large number of records, you may want to know how many records are in the record store. Of course, you can do this with an enumeration and counter, but there's an easier way: simply invoke `getNumRecords` on an open record store. It either returns an integer indicating the number of records currently in the record store, or throws the `RecordStoreNotOpenException` if the record store isn't open.

Listening for Record Store Changes

At times, you may want one application to respond to changes from another application. The record store's `addRecordListener` and `removeRecordListener` methods do just that, letting you register a listener that implements the `RecordListener` interface on an open record store, and deregister a listener, respectively. Once added, a record listener receives the following method invocations whenever a record is added, changed, or deleted, as follows:

- `recordAdded`: The system invokes this method when a record is added.
- `recordChanged`: The system invokes this method when a record is changed.
- `recordDeleted`: The system invokes this method when a record is deleted.

All three of these methods take two parameters: a reference to the record store in which the record is stored, and the record ID of the record being added, changed, or deleted.

Understanding Platform Limitations of Record Stores

Record stores are a good compromise for the variety of mobile devices that can support Java ME. However, they're not perfect—or, more to the point, as you work with them, you learn that their implementation on different devices can be far from perfect. If you're deploying a Java ME application to run on many devices, odds are you will encounter two pitfalls.

First and foremost, just because a device has a certain amount of storage available does *not* guarantee that your MIDlets can use that much storage. The device's software (including the Java applications and native applications such as messaging, contacts/personal information management, web browser, and so forth) shares the persistent store, so the Java ME world only gets a subset of the persistent store in which to store its data. How devices handle managing storage limitations differs from device to device, as some devices may implement per-MIDlet limitations, while others may give the Java AMS a private chunk of the persistent store in which to work. Worse, invoking a `RecordStore`'s `getSizeAvailable` method causes the `RecordStore` to behave differently on different devices; some devices may report the total amount of space left in the currently allocated record store, while other devices may report the total amount of space left in the persistent storage for *all* Java ME record stores.

The second common limitation is a limit on the size of an individual record; this typically stems from the underlying implementation of a given record store. On some devices, writing records that consist of more than some arbitrary maximum size simply fails, throwing an exception, even if there's space in the store for the record. Worse, you can't predict this in advance: there's no way to determine if a specific handset has this limitation, or query the record store implementation to determine what this cap may be. To avoid this, keep your records as short as possible, and if you encounter the problem, break your record up into smaller subrecords and write them individually, chaining them again when you read from the record store.

Putting the Record Store to Work

The WeatherWidget example needs to store the list of locations you entered, as well as the forecast data at each location, so that you don't have to hit the network as you pick locations. The MIDP record store is ideal for this purpose.

WeatherWidget divides its use of the record store up into two classes: the `Location` class, which represents a record in the store once it's loaded into memory, and the `LocationStore` class, which wraps the store in methods more friendly to the application itself. Listing 6-5 shows the `Location` class, which is responsible for storing a location and the weather forecast for a location.

Listing 6-5. *The Implementation of the Location Class*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.rms.*;
import java.io.*;
```

```
public class Location {
    private final static int FIELD_VERSION = 1;
    private final static int FIELD_LOCATION = 2;
    private final static int FIELD_FORECAST = 3;

    public final static int NO_ID = -1;

    private final static int version = 1;
    private String location;
    private String forecast;
    private int recordId;

    /** Creates a new instance of Location */
    public Location(String l, String f) {
        location = l;
        forecast = f;
        recordId = NO_ID;
    }

    public Location(byte[] b) {
        fromBytes(b);
        recordId = NO_ID;
    }


    public Location(byte[] b, int id) {
        fromBytes(b);
        recordId = id;
    }

    public String getLocation() {
        if (location != null) {
            return location;
        } else {
            return "";
        }
    }

    public void setLocation(String l) {
        location = l;
    }
```

```
    public String getForecast() {
        if (forecast != null) {
            return forecast;
        } else {
            return "";
        }
    }

    public void setForecast(String f) {
        forecast = f;
    }

    public int getId() {
        return recordId;
    }

    public void setId(int id) {
        recordId = id;
    }
    public byte[] toBytes() {
        byte[] b;
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);


        // Record format is field-tag, then field for each
        try {
            dos.writeInt(FIELD_VERSION);
            dos.writeInt(version);
            if (location != null) {
                dos.writeInt(FIELD_LOCATION);
                dos.writeUTF(getLocation());
            }
            if (forecast != null)
            {
                dos.writeInt(FIELD_FORECAST);
                dos.writeUTF(getForecast());
            }
        }
        catch( Exception e) {
            return null;
        }
```

```
            // Get the bytes for this item.
            b = baos.toByteArray();
            dos = null;
            baos = null;

            return b;
        }

        public void fromBytes(byte[] b) {
            ByteArrayInputStream bais = new ByteArrayInputStream(b);
            DataInputStream dis = new DataInputStream(bais);

            // Read each tag, then each field
            try
            {
                while(true) {
                    int tag = dis.readInt();
                    switch(tag) {
                        case FIELD_VERSION:
                            // Don't check version; there's only one
                            dis.readInt();
                            break;
                        case FIELD_LOCATION:
                            setLocation(dis.readUTF());
                            break;
                        case FIELD_FORECAST:
                            setForecast(dis.readUTF());
                            break;
                    }
                }
            }
            catch (Exception e) {}

            dis = null;
            bais = null;
        }
    }
```

A location in memory actually has three fields, two of which are visible to its clients:

- `location`: A text string indicating the location for the forecast
- `forecast`: A text string indicating the weather at the location
- `recordId`: An integer indicating the record ID of the record in the record store (if any), or the value `NO_ID` indicating the record has not yet been stored in the record store

The class has accessors and mutators for these three public fields, and it privately keeps a constant field—the record's version number—written with each record into the store. Attaching a version to each record permits you to change the record store implementation without needing to destroy existing records; new code can check the version of each record, select the appropriate code to load a record, and then resave the record in the new format.

The bulk of the methods in the implementation of the `Location` class are getters and setters for each of the public fields. Most interesting are the `toBytes` and `fromBytes` methods, which serialize and deserialize a record to and from an array of bytes, respectively. Although it's unlikely the record format will require much modification, you can choose to use the record-tag approach described previously. This approach is exemplified by `toBytes`, which performs this sequence of steps:

1. It writes a tag indicating that the field version follows the tag.
2. It writes the record version.
3. It writes a tag indicating that the `location` field follows the tag if there's a field to write, followed by the `location` field's contents.
4. It writes a tag indicating that the `forecast` field follows the tag if there's a field to write, followed by the `forecast` field's contents.

The `fromBytes` method works in reverse, except it uses a `while` loop and `switch-case` statements so that it can read the fields of a record in any order.

As shown in Listing 6-6, the `LocationStore` interface provides a proxy to the record store that makes it easy to obtain the list of locations, as well as a forecast for a location. It doesn't wholly abstract the record-store interface—notably, it throws `RecordStoreException` in the event of problems with the underlying store—but it lets you think about accessing locations in the way used by the application rather than solely as records in a store.

Listing 6-6. *The Implementation of the LocationStore Class*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.rms.*;

public class LocationStore {
    private static final String storeName = "wx";
    private RecordStore store = null;

    public LocationStore() {
    }
```

```
    private void openStore() throws RecordStoreException {
        if (store == null) {
            store = RecordStore.openRecordStore(storeName, true);
        }
    }

    private void closeStore() {
        if (store != null) {
            try {
                store.closeRecordStore();
            } catch( Exception ex ) {}
            store = null;
        }
    }

    public String[] getLocationStrings() {
        String[] result = null;
        try {
            openStore();
            result = new String[store.getNumRecords()];
            RecordEnumeration e = store.enumerateRecords(
                null, // No filter
                new RecordComparator () {
                    public int compare(byte[] b1, byte[] b2) {
                        Location r1 = new Location(b1);
                        Location r2 = new Location(b2);
                        if (r1.getLocation().compareTo(➥
r2.getLocation()) == 0) {
                            return RecordComparator.EQUIVALENT;
                        } else
                        if (r1.getLocation().compareTo(➥
r2.getLocation()) < 0) {
                            return RecordComparator.PRECEDES;
                        } else {
                            return RecordComparator.FOLLOWS;
                        }
                    }
                },
                false);
            int i;
```

```
            for (i=0; i<store.getNumRecords(); i++) {
                Location l = new Location(e.nextRecord());
                result[i] = l.getLocation();
                l = null;
            }
            closeStore();
        }
        catch(Exception ex) { closeStore(); }
        return result;
    }

    public Location getLocation( final String location ) throws➥
RecordStoreException {
        openStore();
        Location l = null;
        RecordEnumeration e = store.enumerateRecords(
            new RecordFilter () {
                public boolean matches(byte[] b) {
                    Location r = new Location(b);
                    if (r.getLocation().equalsIgnoreCase(location))➥
return true;
                    else return false;
                }
            },
            null,
            false);

        if (e.hasNextElement()) {
            int id = e.nextRecordId();
            l = new Location(store.getRecord(id), id);
        }
        closeStore();
        return l;
    }

    public void addLocation(Location l) throws RecordStoreException {
        Location existing = getLocation(l.getLocation());
        if (existing!=null) {
            existing.setForecast(l.getForecast());
            updateLocation(existing);
```

```
        } else {
            byte b[] = l.toBytes();
            openStore();
            store.addRecord(b, 0, b.length);
            closeStore();
        }
    }

    public void updateLocation(Location l) throws RecordStoreException {
        int id = l.getId();
        // If it has an id, do an update on that id.
        if (id != Location.NO_ID) {
            byte b[]=l.toBytes();
            openStore();
            store.setRecord(id,b,0,b.length);
        } else {
            // If it doesn't have an id, find it.
            Location target = getLocation(l.getLocation());
            // If there is a record matching this one, update it
            if (target!=null) {
                target.setForecast(l.getForecast());
                updateLocation(target);
            } else {
                // otherwise add this one.
                addLocation(l);
            }
        }
        closeStore();
    }
}
```

The private methods `openStore` and `closeStore` get called a lot. I considered moving `openStore` to the constructor and adding a required cleanup method that the client must call when releasing the interface (remember, the MIDP doesn't support object finalization!), but I decided that explicit finalization would be a burden on the clients of the class. This is subject to change; testing on devices with a lot of locations might dictate a different design. Anyway, these methods are fairly simple: `openStore` opens the store if it isn't already open, rethrowing any exceptions in the event of an error; `closeStore` closes the store, concealing any errors that might occur.

The `getLocationStrings` method returns an array of `Location` items; you can use it for populating the `List` of locations in the main user interface. It uses a simple `RecordEnumeration`, relying on the enumeration to order the resulting enumeration in

increasing alphabetic order. In my anonymous inner `RecordComparator` class, it's a little goofy that `String.compareTo`'s results—negative if the receiver precedes the argument, positive if the receiver succeeds the argument, and zero if they're lexically equivalent—need to be converted to `RecordComparator` values, but that's the way it works. Once you obtain the enumerator, you can simply walk the enumerator and insert each record's location in the appropriate field of the result array. Because this method simulates having the entire list of locations in memory, you can wrap the entire sequence of events in an exception handler.

The `getLocation` method, on the other hand, returns a `Location` instance when provided with the name of a location. It, too, uses a `RecordEnumeration` and looks up the `Location` using the `RecordEnumeration`'s `matches` method. It then uses the resulting enumeration to create a `Location` instance.

■ **Tip** The inner classes for both `locationStrings` and `getLocation` use the same record-parsing code contained within the `Location` class. This is sufficient in this example, where you may have 10 or 20 locations, and likely no more, but it may not be efficient enough to process truly large collections of records, especially on slower handsets. If you're developing an application that must manage a lot of records, consider breaking out field deserialization in such a way as to permit your use of `RecordEnumeration` to only deserialize what it needs to get the job done. Doing so will make your application perform better.

The `addLocation` and `updateLocation` methods are similar, because they must both deal with existing records (you don't want to clutter the record store with duplicate locations of the same place). These methods rely on a `Location` object's `toBytes` method. The `addLocation` method begins by determining if a record for the existing `Location` already exists, and if it does, it uses the `updateLocation` method instead to update that record's contents. Otherwise, it simply gets the bytes corresponding to a serialized version of the indicated `location` instance and adds the record to the database, closing the record store when it finishes.

The `updateLocation` method is a little more complex. It must handle three different cases:

- The `Location` already exists in the database and has an ID (the location was created using the `LocationStore`'s `getLocation` method).
- The `Location` already exists in the store but doesn't have an ID (the location was created programmatically but is a duplicate of one in the store).
- The `Location` does not exist in the store and must be added.

The first case is trivial and uses the store instance's `setRecord` method to update the existing record. The second case is a little more involved and requires that you search the database to get the record ID of the existing record. You do this using the existing `getLocation` method and copying over the data to update from the original `Location` instance. Finally, the third case is also trivial: it just calls out to `addLocation` directly.

Integrating the `Location` and `LocationStore` classes into the WeatherWidget MIDlet is straightforward. Listing 6-7 shows the results.

Listing 6-7. *The Revised WeatherWidget Example Incorporating the Location and LocationStore Classes*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class WeatherWidget extends MIDlet implements CommandListener {
    private Form wxForm;
    private StringItem locationItem;
    private StringItem wxItem;
    private Command exitCommand;
    private Command screenCommand;
    private Command settingCommand;
    private Command okCommand;
    private Command backCommand;
    private List locationList;
    private TextBox locationTextBox;
    private Alert cannotAddLocationAlert;

    String location;
    LocationStore locationStore;

    /** This method initializes the UI of the application.
     */
    private void initialize() {
        locationStore = new LocationStore();
        String[] locations = locationStore.getLocationStrings();
        if (locations.length > 0 ) location = locations[0];

        getDisplay().setCurrent(get_wxForm());
    }
```

```
public void startApp() {
    initialize();
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command command, Displayable displayable) {
    // Insert global pre-action code here
    if (displayable == wxForm) {
        if (command == exitCommand) {
            exitMIDlet();
        } else if (command == settingCommand) {
            getDisplay().setCurrent(get_locationList());
        }
} else if (displayable == locationList) {
    if (command == screenCommand) {
            getDisplay().setCurrent(get_locationTextBox());
    } else if (command == List.SELECT_COMMAND) {
            int index = get_locationList().getSelectedIndex();
            set_location(get_locationList().getString(index));
            getDisplay().setCurrent(get_wxForm());
    } else if (command == backCommand) {
            getDisplay().setCurrent(get_wxForm());
        }
    } else if (displayable == locationTextBox) {
        if (command == backCommand) {
            getDisplay().setCurrent(get_locationList());
        } else if (command == okCommand) {
            add_location(locationTextBox.getString());
            getDisplay().setCurrent(get_locationList());
        }
    } else if (displayable == cannotAddLocationAlert) {
        if (command == backCommand) {
            getDisplay().setCurrent(get_locationList());
        }
    }
}
```

```
public String get_location() {
    if (location == null) {
        location = "";
    }
    return location;
}

public void set_location( String l ) {
    location = l;
    get_wxForm().setTitle(l);
}

public void add_location( String l ) {
    String locations[];
    int i;
    try {
        locationStore.addLocation( new Location( l, "" ));
    } catch (Exception e) {
        getDisplay().setCurrent(get_cannotAddLocationAlert());
    }
    // Refresh the location list lazily.
    locationList = null;
}

public Display getDisplay() {
    return Display.getDisplay(this);
}

/**
 * This method should exit the midlet.
 */
public void exitMIDlet() {
    getDisplay().setCurrent(null);
    destroyApp(true);
    notifyDestroyed();
}

public StringItem get_wxItem() {
    if (wxItem == null) {
        wxItem = new StringItem("Forecast", "Sunny.");
    }
    return wxItem;
}
```

```
    public Form get_wxForm() {
        if (wxForm == null) {
            wxForm = new Form(get_location(), new Item[] {
                get_wxItem()
            });
            wxForm.addCommand(get_exitCommand());
            wxForm.addCommand(get_settingCommand());
            wxForm.setCommandListener(this);
        }
        return wxForm;
    }

    public TextBox get_locationTextBox() {
        if (locationTextBox == null) {
            locationTextBox = new TextBox("Add Location", "", 80, 0);
            locationTextBox.addCommand(get_backCommand());
            locationTextBox.addCommand(get_okCommand());
            locationTextBox.setCommandListener(this);
        }
        return locationTextBox;
    }

   public List get_locationList() {
        if (locationList == null) {
            String[] locations;
            locations = locationStore.getLocationStrings();
            locationList = new List("Where",
                List.IMPLICIT, locations, null);
            locationList.addCommand(get_screenCommand());
            locationList.addCommand(get_backCommand());
            locationList.setCommandListener(this);
        }
        return locationList;
    }

    public Alert get_cannotAddLocationAlert() {
        if (cannotAddLocationAlert == null)
        {
           cannotAddLocationAlert = new Alert("Cannot Add Location");
           cannotAddLocationAlert.setString("An error occurred adding the➥
location you entered. It has not been added.");
           cannotAddLocationAlert.addCommand(get_backCommand());
        }
```

```
        return cannotAddLocationAlert;
    }

    public Command get_settingCommand() {
        if (settingCommand == null) {
            settingCommand = new Command("Settings", Command.OK, 1);
        }
        return settingCommand;
    }

    public Command get_okCommand() {
        if (okCommand == null) {
            okCommand = new Command("OK", Command.OK, 1);
        }
        return okCommand;
    }

    public Command get_exitCommand() {
        if (exitCommand == null) {
            exitCommand = new Command("Exit", Command.EXIT, 1);
        }
        return exitCommand;
    }

    public Command get_screenCommand() {
        if (screenCommand == null) {
            screenCommand = new Command("Add Location", Command.SCREEN, 1);
        }
        return screenCommand;
    }

    public Command get_backCommand() {
        if (backCommand == null) {
            backCommand = new Command("Back", Command.BACK, 1);
        }
        return backCommand;
    }
}
```

Looking at the code and comparing it with the implementation from the previous chapter, you have a minimal number of changes to make. The most important change is the replacement of the `Vector` used to store the list of locations; this is now the job of a `LocationStore` instance, created by the MIDlet's `initialize` method. After opening the

store, the `initialize` method obtains the first location from the store, giving the UI a default location.

The default location returned by `get_location` is now an empty string, consonant with the notion that this will only occur the first time you launch the application before you enter any locations. Updating the list of locations—which `add_location` performs—is no longer a matter of adding an element to a vector, but instead calling the `LocationStore`'s `addLocation` method, with some additional error handling in case the store throws an exception. This error handling relies on the new factory method `get_cannotAddLocationAlert` and the corresponding `Alert`, which informs you that the application couldn't add a location to the list of locations.

The `get_locationList`—a method NetBeans originally created that is the factory method to create the list of locations—now invokes the `LocationStore`'s `getLocationStrings` method instead of converting the `Vector` of locations to an array of `String`s.

Wrapping Up

In this chapter, you learned how to store persistent data using the MIDP's record-store interface. As you now know, a record store is a collection of records, each of which are an ordered collection of bytes. The MIDP implementation of the record-store interface abstracts you from the underlying details of the target, which may store the record store as records in a file, as bytes in flash in a protected area, or as any other medium that meets the standard's requirements. By default, a MIDlet suite's record stores can only be accessed by MIDlets in that suite, although you can export access to other MIDlet suites by specifying an authorization mode when you create the record store.

The `javax.microedition.rms` package contains the record-store classes, which include the `RecordStore` class itself. The `RecordStore` class is responsible for managing record stores as well as inserting, enumerating, updating, and removing record stores. Each record is a collection of bytes that the record store assigns a record ID, which you use to subsequently refer to a record when fetching, updating, or removing the record using the `RecordStore`'s `addRecord`, `setRecord`, `getRecord`, and `deleteRecord` methods. You can also enumerate the records in a store, passing instances of classes that filter and order the enumeration results.

When creating and parsing records in a record store, it helps to use the `java.io` classes `ByteArrayInputStream`, `ByteArrayOutputStream`, `DataInputStream`, and `DataOutputStream`. The `DataInputStream` and `DataOutputStream` let you read and write data primitives, such as integers, floating-point numbers, and strings, to an underlying stream. In conjunction with the `ByteArrayInputStream` and `ByteArrayOutputStream` classes, you can take the resulting streams and convert them to arrays of bytes, which the `RecordStore` class can use immediately as records in the store.

CHAPTER 7

Accessing Files and Other Data

In the last chapter, I showed you how to use the record store—the means of persistent data that is common to *all* MIDP implementations. Sometimes, though, what you really need is raw access to a file on the file system or access to data managed by integrated applications such as built-in contacts managers or date books. Not all devices support this access; those that do implement JSR 75, which defines an optional interface for accessing files on a local file system or personal information management (PIM) data such as that kept by a contacts manager, date book, or to-do program. An implementation supporting JSR 75 may support file system access, PIM data, or both. Moreover, although JSR 75 was developed originally for CLDC devices, it may be found on other Java ME platforms as well. Given its flexibility and growing ubiquity, it's a good interface for you to be familiar with.

In this chapter, I discuss both facets of JSR 75, so you can learn how to access both files and PIM data. I begin with the File Connection Optional Package (FCOP), explaining how it differs from the record store, and I show how it fits in with the Java GCF. Next, I show you the actual APIs you use when interacting with the FCOP. After that, I turn your attention to the PIM package, showing you how the classes in this package fit together to give you an interface that works with the contacts, calendar, and to-do application on a device. In conjunction with the previous chapter, this chapter gives you a firm grasp of how to store data on MIDP devices.

Introducing the FCOP

JSR 75, introduced shortly after CLDC 1.0, solves a problem that hobbled many developers for J2ME, Java ME's immediate predecessor. Java applications were unable to safely interact with integrated applications on a mobile device. The Java community (led by Sun) introduced the PIM interface for specific applications managing PIM data. (I say more about this later in this chapter, in the "Introducing the PIM Package" section.) But for generic applications, or for those applications needing to manage their own files, the FCOP provides the answer.

Devices may or may not have file systems, so you can't assume that all Java ME CLDC devices (let alone all Java ME devices!) support the FCOP as described in JSR 75. If it does exist, though, it plugs into the rest of the Java runtime in a manner similar to that shown in Figure 7-1.

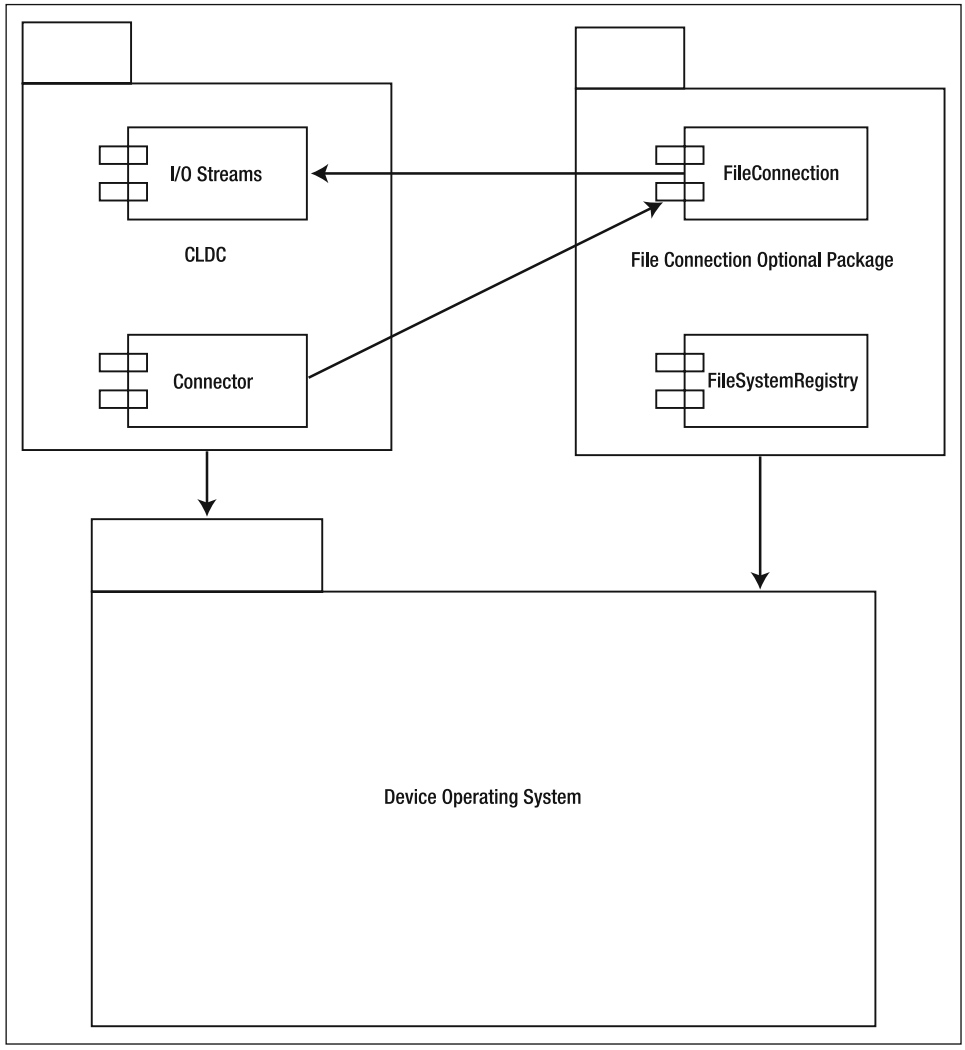


Figure 7-1. *The FCOP and the Java ME package structure*

The key relationship shown in Figure 7-1 is that the `Connector` class defined by the CLDC creates instances of the `FileConnection` interface to manage access to files on the device's file system. Using a `Connector`, you can obtain input and output streams to the file you've referenced, or perform other operations such as removing a file or changing its access mode.

The GCF, which you'll learn about in detail in Chapter 12, provides a unified class hierarchy and means to access local and remote connection resources using URLs and a stream-based interface. The `javax.microedition.io.Connector` class operates as a factory of connections for your application; you pass a URL describing the item you want to open, and the `Connector` class returns an instance of a `Connection` subclass such as `FileConnection`. With the `FileConnection` in hand, you can perform I/O using streams, and you can access the particulars of the file system. You can create new files and directories as well as remove existing files and directories.

A URL referencing a local file on the file system uses the `file://` protocol prefix instead of the `http://` prefix, like this:

```
FileConnection fc =
  (FileConnection) Connector.open("file:///SDC/wx.xml", Connector.READ);
```

Why are there *three* solidus (`/`) characters? The first two separate the file protocol indicator from the path to the file. The region between the second and the third is reserved in URLs for the host name of the object being manipulated; in this case, because the host name is the local host, it's empty.

■ **Caution** It's a common mistake to type out the URL for a file the same way you would a URL for a web resource, using only two slashes. It's an error, however, and one that's hard to catch except by close inspection.

You name files and directories the same way; the URL `file:///SDC/directory` may either be a file or a directory, and it's up to you and the APIs to know the difference. That's important when manipulating directories; as you'll see in a moment, opening a `FileConnection` to `file:///SDC/directory/` and invoking the `FileConnection.mkdir` method to create the directory is an error.

Using the FCOP

The sequence of events when using the FCOP goes something like this:

1. Determine whether or not the device running your application has the FCOP.
2. From the `Connector` class, obtain a `FileConnection` instance representing the file you want to access.
3. Create the file if necessary.
4. Open the file for reading or writing and get an input or output stream, respectively.

5. Perform your input and output using the stream you got from step 4.

6. Close the stream.

7. Close the `FileConnection`.

Of course, you can do several other things, too. You can remove a file or directory, change its access settings, and enumerate the contents of a directory. Finally, you can listen for changes on the file system, which alert you of status changes such as the insertion of a removable storage card like a SanDisk microSD card.

The file system on a device is typically organized as a tree, just as on a FAT32 file system (although the actual nature of the file system might be something totally different). A key difference, however, is that the file system can have *multiple* roots, one for each kind of removable or permanent file system. Thus, `SD1/` might refer to the contents of the first SD card, while `internal` would refer to the internal store. Unfortunately, there's no standard for determining these names; the only way to find out what they mean is to query the system using the change-notification interface provided. (I show you how to do that later, in the section titled "Listening for File System Changes.")

Determining If the FCOP Is Present

Before you begin using the FCOP, you should check to see if it's present. You can do this by interrogating the system for the version of the `FileConnection` interface, as shown in Listing 7-1.

Listing 7-1. *Interrogating the System for the FileConnection Version*

```
String cv = System.getProperty(
            "microedition.io.file.FileConnection.version");
if (cv != null) {
  // Mark that you have the interface available, or just
  // do your file I/O here.
}
```

This check returns a version string such as `"1.0"`, but if it returns anything non-`null`, you know the FCOP is present on the target running your application.

Obtaining a FileConnection Instance

As I indicated previously, you use the `Connection` class to obtain an instance of `FileConnection`, with which you actually perform your file operations. You do this using its `open` method, like this:

```
FileConnection fc =
  (FileConnection) Connector.open("file:///wx.xml", Connector.READ);
```

This method takes two arguments: the URL to the file you wish to open, and the mode in which it should be opened. The mode is one of `Connector.READ`, `Connector.WRITE`, or `Connector.READ_WRITE`.

Not surprisingly, the `open` method can throw exceptions that you must handle, including the following:

- `IllegalArgumentException`: If a parameter is invalid
- `ConnectionNotFoundException`: If the target of the name cannot be found, or if the requested protocol type is not supported
- `IOException`: If some other kind of I/O error occurs
- `SecurityException`: If access to the protocol handler is prohibited

Creating a New File or Directory

Three `FileConnection` interfaces exist to help you manage the creation of files and directories:

- `exists`: Returns a `boolean` indicating whether or not the file name indicated when the `FileConnection` was created exists
- `create`: Creates the named file, assuming you opened it using the `Connector` in `Connector.WRITE` or `Connector.READ_WRITE` mode
- `mkdir`: Creates the named directory, assuming you opened it using the `Connector` in `Connector.WRITE` or `Connector.READ_WRITE` mode

It's important to realize, then, that simply passing a file name to `Connector.open` and invoking it with `Connector.WRITE` or `Connector.READ_WRITE` *doesn't* actually create the file for you!

The creation interfaces are void interfaces, but they can throw `IOException`, `SecurityException`, `IllegalModeException`, or `ConnectionClosedException`; `exists` can throw `SecurityException`, `IllegalModeException`, or `ConnectionClosedException`.

■ **Caution** You cannot create a directory by specifying a trailing solidus (`/`) on the URL for a file and then calling `create`; the trailing solidus will cause `create` to throw an `IOException`. Use `mkdir` instead with a path that does not have a trailing solidus.

Opening a File

Although you've opened a `FileConnection` referring to the file you want to open, you still need to open the file itself to obtain an input or output stream. You do this using one of the following methods:

- `openDataInputStream`: Returns a `DataInputStream` from which you can read the file's contents
- `openInputStream`: Returns an `InputStream` from which you can read the file's contents
- `openDataOutputStream`: Returns a `DataOutputStream` to which you can write data to the file
- `openOutputStream`: Returns an `OutputStream` to which you can write data to the file

Of course, you can only obtain the streams that correspond to the modes in which you opened the `FileConnection`; for example, you couldn't obtain an `OutputStream` if you specified `Connector.READ` to `Connector.open`. Mismatching these will cause the runtime to throw an `IllegalModeException`. You should be prepared to handle the `SecurityException` and `IOException` exceptions as well.

What if you want to write starting somewhere past the first byte of the file? To do this, simply invoke `openOutputStream` with the offset at which you want to begin writing. Be sure your offset is positive, or the runtime will throw an `IllegalArgumentException` your way.

Tweaking File Attributes

The `FileConnection` interface has a host of methods that let you query or tweak attributes of the file or directory to which your `FileConnection` is associated, including the following:

- `canRead`: Returns `true` if your application is permitted to write to the file
- `canWrite`: Returns `true` if your application is permitted to write to the file
- `directorySize`: Returns the number of bytes the directory consumes; you can pass `true` to include the directory's subdirectories in the computation, or `false` to only include the current directory
- `fileSize`: Returns the number of bytes the file consumes
- `getName`: Returns the name of the file, excluding the GCF URL protocol
- `getPath`: Returns the name of the directory containing the file, excluding the GCF URL protocol

- `getURL`: Returns the path and file name in the form used by `Connector` for opening the file (including the GCF URL protocol)
- `isDirectory`: Returns `true` if the specific `FileConnection` refers to a directory
- `isHidden`: Returns `true` if the `FileConnection` refers to a hidden file
- `isOpen`: Returns `true` if the current `FileConnection` is open
- `lastModified`: Returns the last time the file was modified
- `setHidden`: Lets you change the visibility status of a file
- `setReadable`: Lets you change the readable attribute of a file
- `setWritable`: Lets you change the writable attribute of a file
- `truncate`: Lets you truncate a file to a specific length

Of course, all of these can throw an exception, so be prepared to handle any they offer.

Deleting a File or Directory

You can delete a file or directory using the `FileConnection.delete` method, which *immediately* deletes the indicated file or directory. If other streams are associated with the file, they will be flushed and closed automatically; continuing to work with them will result in an `IOException`. The `FileConnection` itself, however, remains open and available for reuse (say, to create a new file with the same name).

Enumerating a Directory's Contents

If you want to obtain a list of the contents of a directory, you can do so using the `FileConnection.list` method, which returns an `Enumeration` of the directory's contents. `FileConnection.list` comes in two flavors: one that takes no arguments and shows all files in a specific directory, and one that takes a wildcard-capable string and a `boolean` indicating whether hidden files should be included in the resulting `Enumeration`.

If you need to manipulate a file or directory returned by the `Enumeration`, you *don't* need to go through the gymnastics of creating a new `FileConnection` by constructing a GCF-compliant URL for the file or directory in question. Instead, you can pass a string from the `Enumeration` directly to the `FileConnection` that opened the `Enumeration` using the `setFileConnection` method. This method resets the `FileConnection` to refer to the file or directory you pass to it.

Listening for File System Changes

Mobile devices are prone to a lot of changes in their environment, and the file system is no exception. Many Java ME devices support removable file systems through memory card slots, such as microSD or miniSD. Consequently, your application needs to be able to detect when the user inserts or removes a card, and you need a way to enumerate the mounted file systems. You can do this by registering a listener to the file system while your application is running; the FCOP provides the `javax.microedition.io.file.FileSystemRegistry` class to do this.

You register a listener using the `FileSystemRegistry.addFileSystemListener` method, passing an instance of a class implementing `javax.microedition.io.file.FileSystemListener`. You override the `rootChanged` method as shown in Listing 7-2.

Listing 7-2. *Overriding the rootChanged Method*

```
FileSystemListener l;

private void startListener() {
    l = new FileSystemListener() {
        public void rootChanged( int state, String rootName ) {
            // Process the change here.
        }
    }
    try {
        FileSystemRegistry.addFileSystemListener( l );
    }
    catch(Exception ex) { /* Handle exception  */ }
}

private void stopListener() {
    if ( l != null ) {
        try {
            FileSystemRegistry.removeFileSystemListener( l );
        }
        catch(Exception ex) { /* Handle exception */ }
    }
}
```

Any time you add or remove a file system, the system invokes your `rootChanged` method with two arguments. The first argument indicates the addition of a file system root if you receive the `FileSystemListener.ROOT_ADDED` value, or the removal of a file system if you receive the `FileSystemListener.ROOT_REMOVED` value. The second argument indicates the name of the root used to access the file system that was added or removed.

The `addFileSystemListener` method may throw one of the following exceptions:

- `SecurityException`: If your application is not permitted to listen for file system changes
- `NullPointerExecption`: If you specify a nonexistent listener

Similarly, the `removeFileSystemListener` method throws a `NullPointerException` if you invoke it with `null`.

■ **Tip** When working in NetBeans, you can simulate the addition and removal of file system roots using the emulator menu. Choose MIDlet ➤ External Events.

Another common use of the `FileSystemRegistry` is to determine which roots are currently mounted. Odds are that you'll want to do this before you first invoke an FCOP class's interface, to help ensure your application is portable between devices. Some devices, like the emulator, ensure that there's a default file system named `root1`, but there's no guarantee that all devices have the same root file names. You can list the root file systems on a device using the `FileSystemRegistry.listRoots` method, which returns an `Enumeration` of `String` items, each a single root file system name. For example, you might write something like what's shown in Listing 7-3.

Listing 7-3. *Enumerating Root File Systems on the Device*

```
Enumeration r = FileSystemRegistry.listRoots();
String cr = null;
while (r.hasMoreElements()) {
   cr = (String) roots.nextElement();
   /* do something with the discovered root */
}
```

Be advised that like the other `FileSystemRegistry` methods, `listRoots` can throw a `SecurityException` if your application isn't permitted to enumerate the root file systems on a device.

Putting the FCOP to Work

Listing 7-4 shows the `LocationStore` class I presented in the previous chapter for storing weather forecasts refactored to use a file instead of the record store.

Listing 7-4. *The LocationStore Implemented with the FCOP*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.io.*;
import javax.microedition.io.file.*;
import java.io.*;
import java.util.*;


public class LocationStore {
    private static final String storeName = "wx";
    private static final String fileURLRoot = "file:///";
    private static String fileUrl;
    Vector locations;
    FileConnection fc;

    public LocationStore() {
        locations = new Vector();
        Enumeration em = FileSystemRegistry.listRoots();
        if (em.hasMoreElements()) {
            fileUrl = fileURLRoot + em.nextElement() + storeName;
        }
        load();
    }

    private void open() {
        if (fc==null) {
            try {
                fc = (FileConnection)Connector.open(fileUrl,
                    Connector.READ_WRITE);
            }
            catch(Exception ex) {}
        }
    }

    private void close() {
        if (fc!=null) {
            try {
                fc.close();
            } catch( Exception ex ) {}
            fc = null;
        }
    }
```

```
public void addLocation(Location l)  {
    locations.addElement( l );
    save();
}

public String[] getLocationStrings() {
    String result[] = new String[locations.size()];
    int i;

    for( i = 0; i < locations.size(); i++ ) {
        result[i] = ((Location)locations.elementAt(i)).getLocation();
    }
    return result;
}

public Location getLocation( final String location ) {
    int i;

    for( i = 0; i < locations.size(); i++ ) {
        Location l = (Location)locations.elementAt(i);
        if (location.equals(l.getLocation())) {
            return l;
        }
    }
    return null;
}

public void updateLocation(Location location) {
    int i;

    for( i = 0; i < locations.size(); i++ ) {
        Location l = (Location)locations.elementAt(i);
        if (location.getLocation().equals(l.getLocation())) {
            l.setForecast(location.getForecast());
            l.setId(location.getId());
        }
    }
    save();
}
```

```
    private void load() {
        try {
            int v;
            byte b[];
            int length;
            Location l;
            DataInputStream dis;

            locations.removeAllElements();
            open();
            dis = fc.openDataInputStream();

            // Read version
            v = dis.readInt();
            // While there are more elements, read them.
            while(true) {
                length = dis.readInt();
                b = new byte[length];
                dis.read( b, 0, length);
                l = new Location(b);
                locations.addElement( l );
            }
        }
        catch(Exception ex) {};
        close();
    }

    private void save() {
        try {
            int i;
            byte[] b;
            DataOutputStream dos;
            open();
            dos = fc.openDataOutputStream();

            // Write version
            dos.writeInt( 1 );
```

```
            for( i = 0; i < locations.size(); i++ ) {
                Location l = (Location)locations.elementAt(i);
                b = l.toBytes();
                dos.writeInt(b.length);
                dos.write(b);
            }
            dos.close();
            close();
        }
        catch(Exception ex) {}
    }
}
```

The first change necessary isn't really one for the FCOP: this `LocationStore` uses a `Vector` to store the list of `Location` instances rather than relying on the file for indexed access. Instead, this change loads all the forecasts into a `Vector`, and each time the `Vector` is changed, it flushes the changes to disk. The constructor creates this `Vector` and then enumerates the mounted file systems, selecting the first (which likely represents the internal store) and constructing a GCF-compliant file name that consists of the `file:///` protocol, the name of the internal file system root, and the name of the store. Finally, it calls `load` to read the records from the file. The `open` and `close` methods open and close the `FileConnection`, respectively.

The `load` method begins by removing any existing elements in the `Vector` and then opening the `FileConnection` and then a `DataInputStream` to the file itself. With the `DataInputStream` open, it reads the version of the file (which will always be `1` in the present implementation) and then loops over the file, reading first the length of a record and then the record as a collection of bytes and converting each collection to a `Location` instance. The `save` method works in the reverse, creating a `DataOutputStream` and writing a version of `1` before iterating over each `Location`, writing first the length and then the binary representation of each `Location`. This approach requires only changes to the `LocationStore` class, not the `Location` class, and does not require any changes to the interface.

■ **Tip** When in doubt, use the `RecordStore` for data storage like this. It's guaranteed to be on every MIDP-compliant device, unlike `FileConnection`, which may not be. The example in this section illustrates how to use the FCOP rather than explain when it should be used.

Integrating the class in Listing 7-4 into the WeatherWidget example is trivial and follows the same basic changes that I describe in the previous chapter. Because the FCOP isn't available on all Java ME devices, the WeatherWidget example uses the record store implementation of the `LocationStore` class.

Introducing the PIM Package

Many Java ME devices act as personal information managers. They contain built-in applications that handle contacts, calendar appointments, and to-do items. Opening these databases to third-party applications can add a lot of value, so JSR 75 defines an optional PIM package that contains classes that abstract access to records of these applications (see Figure 7-2).

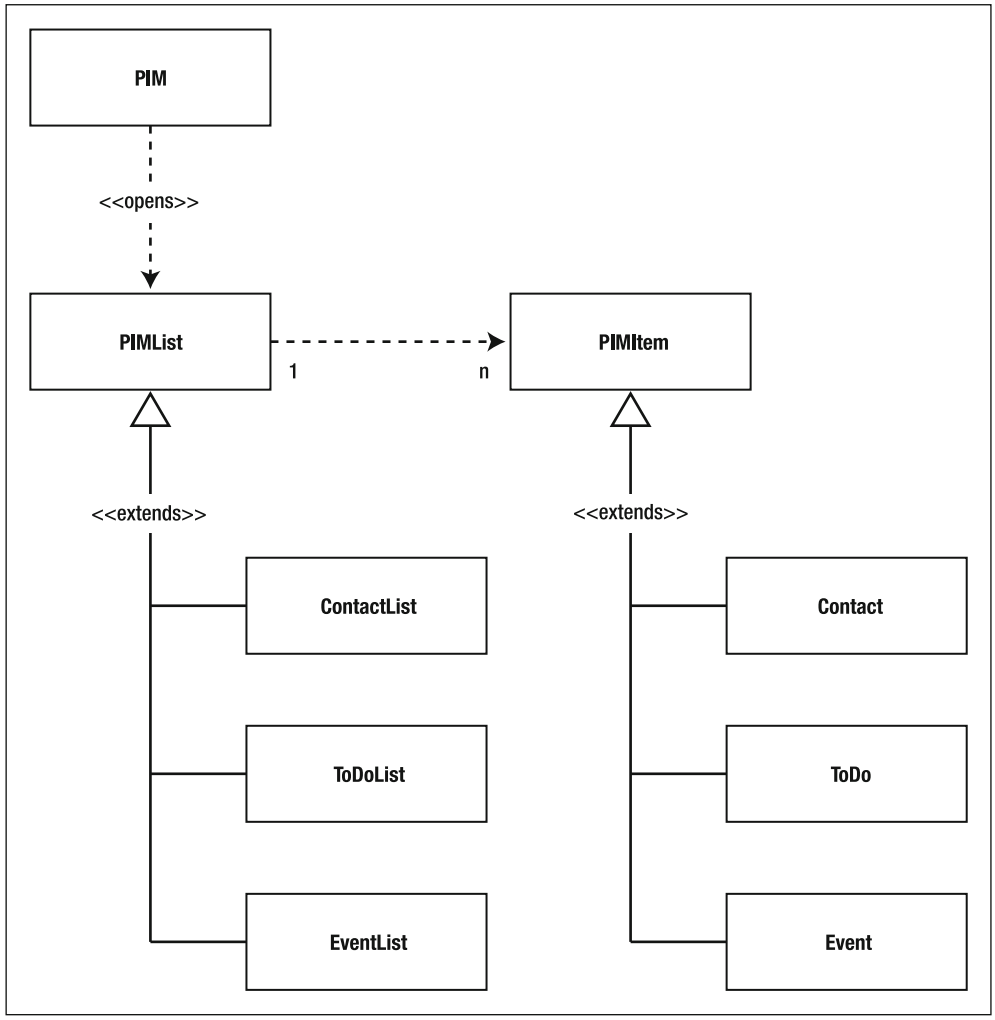


Figure 7-2. *The PIM package classes*

As Figure 7-2 shows, the singleton class `PIM` is responsible for opening instances of `PIMList` subclasses, which include application record stores for the contacts, calendar, and to-do applications. As you'll see, each subclass is represented as an `Enumeration` of

PIMItem objects, and each PIMItem is an interface into a specific item in an application's database. Using the various interfaces in this hierarchy, you can enumerate records in any of these application stores, as well as examine how entries are grouped into categories, create new records, and remove existing records, just as a native application would.

Using the PIM Package

To use the PIM package, you need your code to follow this basic structure:

1. Check the device to ensure the PIM package is available.
2. Open the desired PIM database and obtain a PIMList for the desired database.
3. If you want to read entries from the PIMList, obtain elements from it using one of its Enumeration-yielding interfaces.
4. If you want to create new entries in the PIMList, create a new PIMItem using the PIMList. Then add the data to PIMItem, and finally call commit to commit the changes.
5. If you want to change an entry of the PIMList, obtain the PIMItem corresponding to the item you want to change, and call a method to set the various fields of the item. Then call commit to commit the changes to the item.
6. If you want to remove an item, obtain the PIMItem corresponding to the item to be removed, and then invoke the appropriate remove method on the PIMList, passing the item to be removed.

Unlike the FCOP, which uses different exceptions to flag different errors, the PIM package interfaces throw only a PIMException on an error. The PIMException thrown will have one of the following reason codes available via the getReason method indicating why the exception was thrown:

- FEATURE_NOT_SUPPORTED: If the functionality requested is not supported by the implementation
- GENERAL_ERROR: For general errors
- LIST_CLOSED: If the PIMList you're trying to use is closed already
- LIST_NOT_ACCESSIBLE: If the PIMList you're trying to use is no longer available, perhaps because the underlying database was deleted
- MAX_CATEGORIES_EXCEEDED: If the maximum number of categories for items in the PIMList has been exceeded

- `UPDATE_ERROR`: Indicates an exception where the update could not continue
- `UNSUPPORTED_VERSION`: If the data is not supported by the version of the PIM database

Let's look in more detail at these and the other operations you can perform using the PIM package.

Ensuring the PIM Package Is Available

Before you use the PIM package, you should ensure that it's actually available. You do this using `System.getProperty`, like this:

```
String currentVersion = System.getProperty("microedition.pim.version");
```

This returns the version number of the PIM package, so you can (and should!) check to see if your code is version-compatible with the PIM package that's on the device you're targeting. The call returns `null` if no PIM package is available.

Once you've ensured that the PIM package is available, you must obtain an instance of the PIM singleton that provides access to all of the methods for opening and managing `PIMLists`. You do this using the `PIM.getInstance` method.

Opening a PIM Database

You don't open a PIM database in the same way you would a file, although similar things are probably happening under the hood. Instead, you obtain a `PIMList` instance from the `PIM` class using `PIM.getInstance().openPIMList`. This interface takes two arguments: the type of the list to return (`PIM.CONTACT_LIST`, `PIM.EVENT_LIST`, or `PIM.TODO_LIST`) and an access mode indicating whether the list should be returned as read-only (`PIM.READ_ONLY`), write-only (`PIM.WRITE_ONLY`), or read-write (`PIM.READ_WRITE`).

Some devices have the capability of keeping more than one PIM list of a particular type. Don't confuse this with categories; categories indicate organization within a PIM database, while this feature permits wholly separate lists. You can obtain a list of PIM databases by type by invoking `PIM.getInstance().listPIMLists`, passing one of the PIM list types (`PIM.CONTACT_LIST`, `PIM.EVENT_LIST`, or `PIM.TODO_LIST`). In return, you get an array of `String` instances; each is a name you can pass as an optional third argument to `PIM.getInstance().openPIMList`.

Reading Records from a PIM Database

With the `PIMList` in hand for the PIM database you want to access, you can obtain an `Enumeration` of its contents, returned as `PIMItem` objects. You may use one of three `items` methods to obtain an `Enumeration`:

- An `items` method that takes no arguments and returns an `Enumeration` of *all* items in the database
- An `items` method that takes a `String` and returns an `Enumeration` containing only those records whose `String` fields contain the `String` you passed to `items`
- An `items` method that takes a `PIMItem` template and returns an `Enumeration` containing only those records whose fields contain the fields you specified in the template item (which, of course, must be a `ContactItem`, `EventItem`, or `ToDoItem`, matching the type of `PIMList` you're querying)

These `items` methods can throw one of the following exceptions:

- `IllegalArgumentException`: If the arguments are malformed
- `PIMException`: If an error is manipulating the `PIMList`
- `SecurityException`: If the application is not permitted to access the PIM database

Reading Fields from a PIM Record

The `PIMItem` interface is a generic interface into multiple types of record data: a contact, an event, or a to-do. Consequently, the `PIMItem` uses the notion of *field keys* to access a given datum in a `PIMItem` such as a contact's first name or an event's start time. These fields are defined by well-defined standards such as the vCard specification from the Internet Mail Consortium. Each of the various `PIMItem` interface subclasses defines constant integer accessor keys indicating field types that a `PIMItem` can return in response to an accessor interface. Table 7-1 shows the accessor field keys defined by the PIM package.

Table 7-1. *Accessor Field Keys for the Contact Instances*

Key	Purpose	Type
NAME	The array of the contact's names	String []
ADDR	The array of the contact's primary address fields	String []
EMAIL	The e-mail address of the contact	String
FORMATTED_NAME	The name of the contact	String
NICKNAME	The nickname of the contact	String
NOTE	The note associated with the contact	String
ORG	The contact's organization	String
TEL	The contact's telephone number	String
TITLE	The contact's title	String
UID	The unique ID of the contact	String
URL	The URL stored with the contact	String
PHOTO_URL	A URL to a photo of the contact	String
PUBLIC_KEY_STRING	The public key of the contact as a string	String
BIRTHDAY	The birthday of the contact	Date
REVISION	The last time and date at which the contact was modified	Date
PHOTO	The bytes in the photo of the contact	byte []
PUBLIC_KEY	The public key of the contact as a collection of bytes	byte []
CLASS	Specifies how the contact may be accessed, either as Contact.CLASS_CONFIDENTIAL, Contact.CLASS_PRIVATE, or Contact.CLASS_PUBLIC	int

Table 7-2 shows the accessor field keys for the event instances.

Table 7-2. *Accessor Field Keys for Event Instances*

Key	Purpose	Type
LOCATION	The location of the event	String
NOTE	The note associated with the event	String
SUMMARY	A summary of the event	String
UID	The unique ID of the event	String
END	When the event is scheduled to end	Date
REVISION	The last time and date at which the event was modified	Date

Key	Purpose	Type
START	When the event is scheduled to start	Date
ALARM	The relative time for the event's alarm	int
CLASS	Specifies how the event may be accessed, either as Event.CLASS_CONFIDENTIAL, Event.CLASS_PRIVATE, or Event.CLASS_PUBLIC	int

Table 7-3 shows the accessor field keys for the to-do instances.

Table 7-3. *Accessor Field Keys for To-Do Instances*

Key	Purpose	Type
NOTE	The note for the to-do	String
SUMMARY	A summary of the to-do	String
UID	The unique ID of the to-do	String
PRIORITY	The priority of the to-do from 0 (undefined) to 9, with 1 being the highest priority	int
COMPLETION_DATE	When the to-do was completed	Date
DUE	When the to-do must be completed	Date
REVISION	The last time and date at which the to-do was modified	Date
COMPLETED	Indicates if the to-do was completed	boolean
CLASS	Specifies how the to-do may be accessed, either as ToDo.CLASS_CONFIDENTIAL, ToDo.CLASS_PRIVATE, or ToDo.CLASS_PUBLIC	int

So what to do with all these field constants? There's no guarantee that a specific PIM package implementation will have a specific field in its database, so the first thing to do is to find out which field a particular implementation of the PIM package actually supports. You can do this for a particular key by using `PIMList.getInstance().isSupportedField` or by obtaining an array of all supported field keys using `PIMList.getInstance().getSupportedFields`.

However, to actually read a value in a field, you need to use one of the following `PIMItem` accessor methods in conjunction with the field key:

- `getBinary`: Returns the contents of a binary field as a `byte []` given its field key and the index to the *n*th datum of that type
- `getDate`: Returns the contents of a date field as a `Date` given its field key and the index to the *n*th datum of that type

- `getInt`: Returns the contents of an integer field as an `int` given its field key and the index to the *n*th datum of that type
- `getString`: Returns the contents of a string field as a `String` given its field key and the index to the *n*th datum of that type
- `getBoolean`: Returns the contents of a `boolean` field as a `boolean` given its field key and the index to the *n*th datum of that type
- `getStringArray`: Returns the contents of a string array field as a `String []` given its field key and the index to the *n*th datum of that type

For example, the code shown in Listing 7-5 returns the summary of a to-do item.

Listing 7-5. *Returning the Summary of a To-Do Item*

```
ToDoList list;
ToDo item;
// fetch item using an enumeration from PIMList
if (list.isSupportedField(ToDo.SUMMARY)) {
    String summary = item.getString( ToDo.SUMMARY, 0 );
    // Do something with the summary
}
```

Each field can contain multiple entries for the same type, which is why you pass an index to each of these accessor methods. For example, to enumerate over all phone numbers in a contact, you might write the code shown in Listing 7-6.

Listing 7-6. *Enumerating Over Record Fields in a Contact*

```
ContactList list;
Contact item;
int i=0;

// fetch item using an enumeration from PIMList
if (list.isSupportedField(Contact.TEL)) {
    String phone;
    phone = item.getString(Contact.TEL, i);
    while(phone!=null) {
        // Do something with the phone here.
        i++;
        phone = item.getString(Contact.TEL, i);
    }
}
```

A datum for a field with multiple values such as `Contact.TEL` may have an attribute indicating what *kind* it is. For example, a contact may have multiple telephone numbers, one each for home, mobile, office, and fax phones. While you use the field index to obtain each of these numbers, you use the attribute of a given field to determine *which* number is which. You do this using the `getAttributes` method of a `PIMItem` instance, which returns a bit mask of attributes for the given field and index, like so:

```
int attrs = item.getAttributes( Contact.TEL, i );
```

As I write this, attributes are only used for `Contact` items, and Table 7-4 shows the list of attributes for `Contact` items. Of course, this may change in future versions of the PIM package.

Table 7-4. *Attributes of Contact Fields*

Attribute	Purpose
`ATTR_ASST`	Field datum related to an administrative assistant
`ATTR_AUTO`	Field datum related to auto
`ATTR_FAX`	Field datum related to facsimile
`ATTR_HOME`	Field datum related to home
`ATTR_MOBILE`	Field datum related to mobile
`ATTR_OTHER`	Field datum related to other information
`ATTR_PAGER`	Field datum related to pager
`ATTR_PREFERRED`	Field datum related to preferred contact (may only be on one datum)
`ATTR_SMS`	Field datum related to SMS
`ATTR_WORK`	Field datum related to work

Two fields—`Contact.NAME` and `Contact.ADDR`—return an array of strings. This is because different implementations may have differing numbers of fields for a single name or address. For example, does an address consist of one line for the street and one for the suite or apartment, or two? Consequently, you can access each portion of these fields as a single string within the returned array of strings. The `Contact` class provides constants that let you index these arrays to obtain specific parts of a field, such as the city, street, or postal code of an address. Table 7-5 shows the indexes you can use when accessing elements of these `String` arrays. You can also determine the size of these arrays by invoking a `ContactList`'s `stringArraySize`, passing either `Contact.NAME` or `Contact.ADDR` to obtain the maximum number of permissible elements in either array.

Table 7-5. *Indexes into Contact Field String Arrays*

Attribute	Purpose
ADDR_COUNTRY	Country part of address
ADDR_EXTRA	Extra part of address
ADDR_LOCALITY	Locality part of address
ADDR_POBOX	Post office box part of address
ADDR_POSTALCODE	Postal code part of address
ADDR_REGION	Regional part of address
ADDR_STREET	Street part of address
NAME_FAMILY	Family name portion of name
NAME_GIVEN	Given name portion of name
NAME_OTHER	Other portion of name
NAME_PREFIX	Prefix portion of name (i.e., Dr.)
NAME_SUFFIX	Suffix portion of name (i.e., Jr.)

Modifying a PIM Record

Once you have a `PIMItem` in hand, you can change it. There are three kinds of changes you might want to make:

- Add a datum to an existing field (e.g., add a new phone number to a contact with existing phone numbers)
- Replace a datum within an existing field (e.g., replace an existing phone number with new information)
- Remove a field entirely

In the first case, you use the `PIMItem`'s adder methods; in the second and third cases, you use the `PIMItem`'s setter methods to do this.

The following adder methods are available:

- `addBinary`: Adds a new binary (`byte[]`) field datum
- `addDate`: Adds a new `Date` field datum
- `addInt`: Adds a new integer (`int`) field datum

- `addString`: Adds a new `String` field datum
- `addBoolean`: Adds a new `boolean` field datum
- `addStringArray`: Adds a new field datum consisting of an array of `String`s

All of these methods take the field, attributes, and field datum you want to add to an existing field.

The following setter methods are available:

- `setBinary`: Sets the value of a binary field
- `setDate`: Sets the value of a `Date` field
- `setInt`: Sets the value of an integer field
- `setString`: Sets the value of a `String` field
- `setBoolean`: Sets the value of a `boolean` field
- `setStringArray`: Sets the value of a field consisting of an array of `String`s

When you invoke one of these methods, you pass the field you wish to add or replace, the index, any attributes of the item, and the data to set.

Once you finish changing a `PIMItem`, you *must* invoke its `commit` method to write the changes back to the PIM package's record store.

Adding a PIM Record

Adding a new record is a three-step process:

1. Call the factory method in the `PIMList` interface subclass to create an empty record of the desired type.
2. Invoke the various adder methods on the new record to populate it with the data.
3. Call `commit` to flush your changes to the PIM package's record store.

The `PIMList` subclasses each define a factory method (`createContact`, `createEvent`, or `createToDo`), which you must use to create an empty record in the application's record store. Once you do, you can fill out the fields of the resulting record and then simply flush it to the store. Listing 7-7 shows how to create a new contact (omitting the obligatory exception handling).

Listing 7-7. *Creating a New Contact*

```
ContactList list = (ContactList)
        PIM.getInstance().openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE);
Contact c = list.createContact();
    String [] name = new String[list.stringArraySize(Contact.NAME)];
    name[Contact.NAME_GIVEN] = "Ray";
    name[Contact.NAME_FAMILY] = "Rischpater";
    c.addStringArray(Contact.NAME,Contact.ATTR_NONE,name);
    c.commit();
```

Removing a PIM Entry

To remove a PIM entry, you must first have a reference to the corresponding `PIMItem` item; you can get this reference with any of the enumeration methods described in the previous section, "Reading Records from a PIM Database." With the entry in hand, call the appropriate `remove` method for the type of `PIMList` you're managing. You call either `removeContact`, `removeEvent`, or `removeToDo`, then pass the `PIMEntry` instance corresponding to the item to be removed.

Note that there's no method to remove all items in a list; to do this, you need to enumerate through the items in the list and remove each in turn.

Managing PIM Database Categories

Some devices support the ability to group items into categories, such as work and home contacts. You can enumerate a `PIMList`'s categories using the `getCategories` method, which returns a `String` array containing category names. If the result is a zero-length list, either categories are not supported or there are no categories defined for the `PIMList`. You can also test to see if a given string is used as a category name by invoking `isCategory` and passing a string containing the name of the category you'd like to test.

You can add and remove categories using the `addCategory` and `deleteCategory` methods. When adding a category, the category is only added if it does not exist already; you can't have two categories with the same name. This determination is made in a case-sensitive way, although the underlying implementation may not be case sensitive. In a similar vein, `deleteCategory` removes the category you name, although you must also pass a `boolean` indicating whether you'd like to delete the items that were left unassigned to any category as a result of removing that category. You can also rename a category, using the `renameCategory` method; pass the current category name and the new category name (in that order).

Categories are typically used to limit the number of items that show in a list; you can do the same programmatically using the `PIMList`'s `itemsByCategory` method, passing the name of a category. In turn, you receive an `Enumeration` (possibly empty) of the `PIMItems` in the category you specify.

Putting the PIM Package to Work

Listing 7-8 shows a sample application that uses the PIM package. It performs two functions: on launch, it inserts two entries into the contacts database, and when you press a soft key, it lists the names of all the contacts in the contacts database.

■ **Note** When running this sample application in the Sun emulator or as an unsigned application on a handset, you'll receive many requests by the AMS asking you for permission to perform each operation. This is because the MIDlet requires privilege to use the PIM package in order to execute, and it must be packaged as a signed application to avoid prompting the user before that privilege is used.

Listing 7-8. *The PIM Package at Work*

```
package com.apress.rischpater;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;
import javax.microedition.pim.*;

public class PIMExample extends MIDlet implements CommandListener {
    public PIMExample() {
        try {
            verifyPIMSupport();
            seed();
        }
        catch (Exception ex) {
            Form mForm = new Form("Exception");
            mForm.append(new StringItem(null, ex.toString()));
            Command mExitCommand = new Command("Exit", Command.EXIT, 0);
            mForm.addCommand(mExitCommand);
            mForm.setCommandListener(this);
            return;
        }
    }
```

```
    private Form helloForm;
    private StringItem helloStringItem;
    private Command exitCommand;
    private List listContacts;
    private Command okCommand1;

    private void initialize() {
        getDisplay().setCurrent(get_helloForm());
    }

    public void commandAction(Command command, Displayable displayable) {
        if (displayable == helloForm) {
            if (command == exitCommand) {
                exitMIDlet();
            } else if (command == okCommand1) {
                getDisplay().setCurrent(get_listContacts());
            }
        }
    }

    public Display getDisplay() {
        return Display.getDisplay(this);
    }

   public void exitMIDlet() {
        getDisplay().setCurrent(null);
        destroyApp(true);
        notifyDestroyed();
    }

    public Form get_helloForm() {
        if (helloForm == null) {
            helloForm = new Form(null, new Item[] {get_helloStringItem()});
            helloForm.addCommand(get_exitCommand());
            helloForm.addCommand(get_okCommand1());
            helloForm.setCommandListener(this);
        }
        return helloForm;
    }
```

```
public StringItem get_helloStringItem() {
    if (helloStringItem == null) {
        helloStringItem = new StringItem("", "");
    }
    return helloStringItem;
}

public Command get_exitCommand() {
    if (exitCommand == null) {
        exitCommand = new Command("Exit", Command.EXIT, 1);
    }
    return exitCommand;
}

public List get_listContacts() {
    if (listContacts == null) {
        listContacts = new List(null, Choice.IMPLICIT,
            new String[0], new Image[0]);
        listContacts.setCommandListener(this);
        listContacts.setSelectedFlags(new boolean[0]);
        ContactLoaderThread t = new ContactLoaderThread(listContacts);
        t.start();
    }
    return listContacts;
}

public Command get_okCommand1() {
    if (okCommand1 == null) {
        okCommand1 = new Command("Ok", Command.OK, 1);
    }
    return okCommand1;
}

public void startApp() {
    initialize();
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}
```

```
public void verifyPIMSupport() throws IOException {
    String version = "";
    version = System.getProperty("microedition.pim.version");
    if (version != null) {
      if (!version.equals("1.0"))
        throw new IOException("Package is not version 1.0.");
    }
    else
      throw new IOException("PIM optional package is not available.");
}

private ContactList list = null;

private void seed() throws PIMException {
    try {
        PIM pim = PIM.getInstance();
        list = (ContactList)
            pim.openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE);
    }
    catch (PIMException ex) { /* Contact list is not supported. */ }
    addContact(list, "Ray", "Rischpater", "1234 High Street",
        "Los Gatos", "USA", "95030");
    addContact(list, "John", "Doe", "1111 Bear Road",
        "Mariposa", "USA", "9????");
    if (list != null)
        list.close();
    list = null;
}

private void addContact( ContactList list, String firstName,
                         String lastName, String street, String city,
                         String country, String postalcode)
throws PIMException {
    Contact c = list.createContact();
    String [] name = new String[list.stringArraySize(Contact.NAME)];
    name[Contact.NAME_GIVEN] = firstName;
    name[Contact.NAME_FAMILY] = lastName;
    c.addStringArray(Contact.NAME, Contact.ATTR_NONE, name);
    String [] addr = new String[list.stringArraySize(Contact.ADDR)];
    addr[Contact.ADDR_STREET] = street;
    addr[Contact.ADDR_LOCALITY] = city;
```

```
            addr[Contact.ADDR_COUNTRY] = country;
            addr[Contact.ADDR_POSTALCODE] = street;
            c.addStringArray(Contact.ADDR, Contact.ATTR_NONE, addr);
            c.commit();
        }
}
```

Most of this is boilerplate autogenerated by NetBeans; the stuff you're interested in is invoked by the constructor: `verifyPIMSupport`, `addContact`, `seed`, and `get_listContacts`. The constructor simply verifies support for the PIM package and invokes `seed` to add new contacts to the database, creating and showing an error form if either method throws an exception. Speaking of throwing exceptions, `verifyPIMSupport` does just that if it can't find the PIM package, or if the PIM package supported by the device isn't the version that the application expects (version 1.0).

The `seed` method creates two contacts by opening the `PIM` instance and obtaining a `ContactList` to which it adds each of two statically defined contacts. This is handy code to have around, because with it you can test your own PIM code directly in the emulator as necessary. The `addContact` method uses the `ContactList`'s `createContact` method to create an empty contact, and then adds the fields to the contact using the `Contact.NAME` and `Contact.ADDR` fields.

The `get_listContacts` method is interesting, if only because it uses a separate thread to read the contacts from the database. While you can do this synchronously, I find that multithreading this operation leads to better behavior in the emulator; Listing 7-9 shows the resulting `ContactLoaderThread` class.

Listing 7-9. *Populating a List with Contacts Using the ContactLoaderThread*

```
package com.apress.rischpater;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;
import javax.microedition.pim.*;

public class ContactLoaderThread extends Thread {
    List list;

    public ContactLoaderThread(List l) {
        list = l;
    }
```

```
    public void run() {
        try {
            PIM pim = PIM.getInstance();
            ContactList contList = (ContactList)
                pim.openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY);
            Enumeration contacts = contList.items();
            while(contacts.hasMoreElements()) {
                Contact c = (Contact) contacts.nextElement();
                String [] nameValues = c.getStringArray( Contact.NAME, 0);
                String firstName = nameValues[Contact.NAME_GIVEN];
                String lastName = nameValues[Contact.NAME_FAMILY];
                list.append(lastName + ", " + firstName, null);
            }
        }
        catch(Exception ex) {}
    }
}
```

The class in Listing 7-9 is simple, too—its run method simply iterates over each `Contact` in the open `ContactList`, getting the first and last names of each item and appending them to the `List` instance passed by the calling thread. The code expressly catches exceptions but does nothing with the exceptions; a real-world application would probably navigate to yet another `Form` to show an error.

Understanding the Role Code Signing and Verification Can Play

As you might imagine, a rogue application can do some real damage using the interfaces described in this chapter. While not all Java ME implementations open up access to the entire file system and applications suites—device vendors are free to expose only a portion of the file system, keeping protected files such as those representing system or network access settings hidden from *all* Java ME applications—the file system space accessible via the FCOP is typically shared among multiple Java ME applications as well as potential system applications. A rogue application (either accidental or by design) consuming file system space willy-nilly or deleting files or PIM records belonging to other applications could cause no end of grief for a device user.

To help protect against this, most device manufacturers and carriers only support access to the interfaces in JSR 75 within applications that have been signed in some way (see the "Marketing and Selling Your Application" section in Chapter 1 and the "Packaging and Executing CLDC/MIDP Applications" section in Chapter 3). Code signing ensures that only those applications trusted by the operator (and by extension,

the user) have access to handset data. Obtaining this trust is typically a matter of third-party certification prior to distribution. While not perfect—of course, it's possible that even a well-crafted application might have a defect that results in improper data access despite passing certification tests—such a scenario is far less likely than in an untrusted computing environment.

Typically, devices distributed by a network operator impose signing requirements for a number of APIs, including file system (and PIM) access. The signing requirements are often hierarchical, and for full access, you may need to distribute your application on the network operator's deck, requiring a business arrangement between your firm and the network operator. Consequently, even if a device supports these APIs, they may not be commercially available to your users on some devices.

Wrapping Up

Although not available on every Java ME device, JSR 75 defines two packages for accessing existing data outside the Java sandbox. The FCOP, building atop the GCF, lets you access and manage files on internal and removable file systems. You access files and directories for reading and writing using the `Connector` class, and then use the resulting `FileConnection` instance to open the file or perform directory-level actions such as creating new files or directories or removing existing files or directories. Using the `FileConnection`, you can also obtain `InputStream` and `OutputStream` instances for your file, letting you read and write data from files on the file system. You can also listen for changes to the file system that notify your application when removable media is inserted or removed, letting your application respond to file system change events.

The PIM package, also defined by JSR 75, provides a set of abstract interfaces that let you access contact, event, and to-do databases. Using the `PIM` class, you access a singleton that then lets you open PIM databases—represented by the `PIMList` interface—and enumerate native database records in each of these applications. The resulting records are represented as `PIMItem` objects, which consist of fields you access by enumerated key via getters and setters.

CHAPTER 8

Using the Java Mobile Game API

Gaming—especially casual gaming—has become big business for the mobile market. Today's high-end cell phones rival portable game consoles in terms of both raw processing horsepower and display fidelity; even low-end consumer phones can support many kinds of games, as they often have computational characteristics equivalent to a PC from only a handful of years ago. Java ME rises to the challenge quite handily, offering a robust API for developing 2D games as part of MIDP 2.0.

In this chapter, I show you the classes that make up the Java Mobile Game API found in MIDP 2.0 and beyond. I show you how to manage events and drawing, including how to optimize your code to poll for keystrokes. I explain how to structure the graphics in your game around layers using Java ME's support for layers, tilings, and sprites, and how to manage your game in the context of a single primary game loop. Finally, I close with a brief example that ties all this together, showing you how to create a simple game animation using the API.

Looking Inside the Mobile Game API

While you *could* write games using the Java ME classes you've learned about so far—many developers did, back in the days of MIDP 1.0—doing so is a lot of work. Typically, you'd subclass `Canvas`, use double buffering to avoid drawing flicker, and work carefully with one or more threads to control the various aspects of your game. You'd likely want to build your own display manager that would let you create game levels as collections of tiles, perhaps adding simple sprite animation. Fortunately, you get all this for free in MIDP 2.0 and beyond, and a few other things besides (see Figure 8-1).

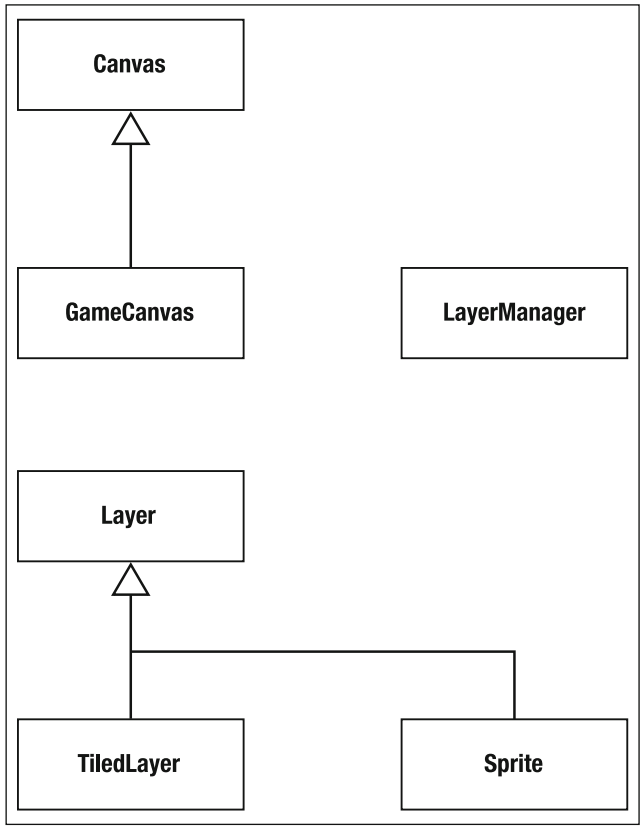


Figure 8-1. *The classes in the Mobile Game API included in Java MIDP 2.0 and beyond*

Five classes make up the Mobile Game API:

- `GameCanvas`: A subclass of `Canvas` used for accepting keystroke events and visually containing your game
- `LayerManager`: A class responsible for presenting visible layers on the screen and maintaining their Z-order
- `Layer`: An abstract class describing how a layer to be displayed on the screen should behave
- `TiledLayer`: An implementation of `Layer` suited for displaying large bitmaps made up of regular repeating regions called *tiles*
- `Sprite`: An implementation of `Layer` suited for animation of a set of bitmaps according to an object's state

The `javax.microedition.lcdui.game` package contains all of these classes.

Managing Events and Drawing

The `GameCanvas` class provides a basis for your game's visual display, including all of the capabilities of the `Canvas` class (see Chapter 5, especially the section "Working with the Canvas and Custom Items"). The class provides interfaces for managing the following events:

- *Event handling*: You can continue to receive key and pointer events to control your application. As you'll soon see, you can also poll for key events, which is handy if you're within your game's control loop.

- *Command handling*: `GameCanvas` ultimately inherits from `Displayable`, so you inherit all the methods pertaining to the `Command` infrastructure, including `addCommand`, `removeCommand`, and `setCommandListener`. In practice, you won't use this often, as raw keystrokes become the usual means for controlling a game.

- *Drawing*: Your `GameCanvas` can implement `paint`, which is responsible for painting the display. However, as you'll soon see, it may be more advantageous to do your painting within your game's control loop instead.

The `GameCanvas` has several additional features. First, `GameCanvas` has its own screen buffer associated with it. `GameCanvas` subclasses are assured that the only thing that can draw to a `GameCanvas` buffer is the `Graphics` objects obtained directly from that canvas, without interference from other controls, MIDlets, or the system. With this buffer comes added responsibility; as graphics buffers can be large, you should be conservative when you allocate `GameCanvas` objects, and reuse them whenever possible.

Second, when you create a `GameCanvas` subclass, you have the option of indicating whether you want the system to dispatch key events to the canvas, or whether you want to poll the system for key states. This is a key way to optimize your application; if your application has a central control loop, it is usually faster to poll for key state changes during your event loop instead of responding to key events and updating the game state, because it saves you the overhead of a method dispatch and the complexity of tracking control state in your event handler and responding to changes in state in your control loop.

Finally, because your application owns its display buffer, it can draw to the buffer at any time. Now, this isn't a call to generating graphics operations willy-nilly; such a style can hurt game performance. However, it does invite a style of programming more familiar to game programmers of yore, in which a central control loop running on its own thread is responsible for updating game state and redrawing the display.

Polling for Keystrokes

Polling for keystrokes is easy: simply use the `getKeyStates` method. This method returns a bitmask of the pressed key values shown defined in the `GameCanvas` class in Table 8-1; you can simply use `&` to mask off the flag for the key whose state you seek. Not all of these keys are available on all devices; for example, many phones don't include hardware to generate the `GAME_A`–`GAME_D` events.

Table 8-1. *Keys Available via getKeyStates*

Value	Typical Action
`KEY_NUM0`–`KEY_NUM9`	Numeric keypad keys 0–9
`KEY_POUND`	# key
`KEY_STAR`	* key
`LEFT`	Left directional key
`RIGHT`	Right directional key
`UP`	Up directional key
`DOWN`	Down directional key
`FIRE`	OK or center directional key
`GAME_A`, `GAME_B`, `GAME_C`, `GAME_D`	Custom game keys

■ **Caution** As I point out in Chapter 5, not all devices provide all key codes, and some devices map multiple key codes to the same key. When writing your application, be sure to abstract the key code from the behavior it embodies using a lookup table or `switch-case` statement so you can accommodate this.

Of course, if you're polling for key state, you don't want to receive key events as well, so you should pass `true` to your `GameCanvas`' `super` invocation to indicate that event handling should be disabled. For example, in my game, I might write something like what's shown in Listing 8-1.

Listing 8-1. *Polling for Key States*

```
public class MyGameCanvas
    extends GameCanvas {
    Sprite cat;

    public MyGameCanvas () {
        super(true);
        /* Do other setup here */
    }

    private void moveCat() {
        int keyStates = getKeyStates();
        int x, y;
        x = cat.getX();
        y = cat.getY();
        if ((keyStates & LEFT_PRESSED) != 0) {
            x-=cat.getWidth()/4;
        }
        if ((keyStates & RIGHT_PRESSED) != 0) {
            x+=cat.getWidth()/4;
        }
        if ((keyStates & UP_PRESSED) != 0) {
            y-=cat.getHeight()/4;
        }
        if ((keyStates & DOWN_PRESSED) != 0) {
            y+=cat.getHeight(){/4;
        }
        cat.setPosition(x,y);
    }
    /* The rest of the canvas's implementation */
}
```

Managing Game Execution

Polling for events is all well and good, but just *where* do you poll for events? You poll in the game's control loop, which is a separate thread that you can usually implement in the same class as your `GameCanvas`, as shown in Listing 8-2.

Listing 8-2. *Implementing the Game's Control Loop in GameCanvas*

```
public class MyGameCanvas
    extends GameCanvas
    implements Runnable {
    Sprite cat;

    public MyGameCanvas () {
        super(true);
        /* Do other setup here */
    }

    private void moveCat() {
        int keyStates = getKeyStates();
        int x, y;
        x = cat.getX();
        y = cat.getY();
        if ((keyStates & LEFT_PRESSED) != 0) {
            x-=cat.getWidth()/4;
        }
        if ((keyStates & RIGHT_PRESSED) != 0) {
            x+=cat.getWidth()/4;
        }
        if ((keyStates & UP_PRESSED) != 0) {
            y-=cat.getHeight()/4;
        }
        if ((keyStates & DOWN_PRESSED) != 0) {
            y+=cat.getHeight()/4;
        }
        cat.setPosition(x,y);
    }


    private final int DELAYMS = 100;

    public void start() {
        Thread t = new Thread(this);
        t.start();
    }
```

```
    public void run() {
        Graphics g = getGraphics();
        while(true) {
            updateGameState(); // move computer controlled characters
            moveCat();         // poll for events, move main character
            doPaint();         // paint screen

            try {
                Thread.sleep(DELAYMS);
            }
            catch(InterruptedException ex) {}
        }
    }
}
```

In this more-than-pseudocode, less-than-a-game, the polling occurs in the `moveCat` method I showed you in the previous section. Once the thread is started, the thread updates the game's characters, polls the keys, and repaints the screen every `DELAYMS` (100 milliseconds). The `updateGameState` and `doPaint` methods aren't defined yet, of course—more on them as we continue to explore the API.

Tying Your GameCanvas to Your MIDlet

As you remember from the discussion of the `Canvas` class in Chapter 5, it and its subclasses implement `Displayable`, meaning you can set the `GameCanvas` subclass to be the active `Displayable` using the `Display`'s `setCurrent` method, as shown in Listing 8-3.

Listing 8-3. *Setting the GameCanvas to Be the Active Displayable*

```
public class GameCanvasSampleMIDlet extends MIDlet {
    private MyGameCanvas canvas;

    public GameCanvasSampleMIDlet () {
    }

    private void initialize() {
        canvas = new MyGameCanvas();
        getDisplay().setCurrent(canvas);
        canvas.start();
    }
```

```
    public void exitMIDlet() {
        canvas=null;
        getDisplay().setCurrent(null);
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp() {
        initialize();
    }

    public Display getDisplay() {
        return Display.getDisplay(this);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

Here, the MIDlet's entry point calls `initialize`, which in turn creates the `MyGameCanvas` and sets the display's `Displayable` to the new `MyGameCanvas` instance before starting its game loop.

This—creating a `GameCanvas` subclass that implements `Runnable` for its game loop, setting the `GameCanvas` to be the current `Displayable`, and starting the game loop—is at the heart of any MIDP 2.0–based game application.

Layering Visual Elements

While the introduction of the `GameCanvas` with its support for key polling, double-buffered graphics, and the game loop is an important advancement in game programming for Java ME, that's only half the story. The `Layer` class hierarchy and the related `LayerManager` greatly simplify how you handle graphics in your game.

A `Layer` is an abstract class representing a visible element of a game. `Layer`s must know how to paint themselves, as well as track their position and visibility. You can perform the following operations on a `Layer`:

- Obtain its position on the canvas using the `getX` and `getY` methods.
- Obtain its width and height using the `getWidth` and `getHeight` methods.

- Determine (and set) whether or not the `Layer` is visible using the `isVisible` and `setVisible` methods.

- Set its position (either by offset or absolute positioning relative to the `Graphics` object responsible for painting the object) using the `move` and `setPosition` methods.

The `Layer` itself is interesting, but it's not as interesting as the `TiledLayer` and `Sprite` subclasses, which provide concrete implementations of `Layer`.

You use the `TiledLayer` to present large images consisting of small regular repeating bitmaps, or tiles. The `TiledLayer` is good for providing game backgrounds as well as intermediate layers. A `TiledLayer` instance divides a large region into *cells*, each of which is assigned a *tile*, a subimage from within the `TiledLayer`'s `Image` instance.

While you use the `TiledLayer` for large static objects, you use the `Sprite` for smaller animated objects. Like a `TiledLayer`, a `Sprite` takes an image that gets divided up into smaller images, called *frames*. Unlike a `TiledLayer`, a `Sprite` displays a *single* frame at its location, letting you choose which frame should be displayed. `Sprite` instances can transform the frame they're displaying through rotation or mirror flipping, giving you the ability to specify various appearances of an animated item using only a handful of frames.

The various layers are tied together using the `LayerManager` class, which maintains an ordered list of the `Layers` it must draw. This ordered list provides Z-ordering for the frames; the first item in the list (at index `0`) is closest to the user. Using the methods of the `LayerManager`, you can add and remove layers from this list, as well as repaint the individual `Layers`.

Managing Layers

The `LayerManager`'s primary responsibility is to help you keep the `Layers` in your game organized. To do this, the `LayerManager` encapsulates a list of `Layers` and provides a view window that defines the size of the visible region of the `Layers` and the visible region's position relative to the `LayerManager`'s coordinate system. By panning the view window, you can pan the display across a collection of layers, creating a viewable window that scrolls in any direction over the game world. You do this using the `setViewWindow` method, passing the bounds (e.g., left, top, width, height) of the view window. For example, to scroll right, simply move the view window to the right (increment the `x` coordinate when invoking `setViewWindow`). Often, you want to coordinate this behavior with the movement of the user character's `Sprite`, so that the game world visible corresponds to the immediate surroundings of the user.

The majority of the `LayerManager` methods correspond to actions you can take on the `LayerManager`'s list of `Layers`, as follows:

- `append`: Takes a `Layer` and places it at the end of the `LayerManager`'s list
- `insert`: Takes a `Layer` and an index and inserts the `Layer` at that index in the `LayerManager`'s list, sliding subsequent `Layer`s behind the inserted `Layer`
- `getLayerAt`: Takes an index and returns the `Layer` at that index
- `remove`: Takes a `Layer` and removes that `Layer` from the `LayerManager`'s list
- `getSize`: Returns the number of `Layer`s in the `LayerManager`'s list

■ **Caution** Remember that the order in which you add items to the `LayerManager`'s list determines the Z-order for drawing, and the front-most item is at index `0`—the first item you add to the list! It's not uncommon to build your list of `Layer`s, only to find that the background is the only thing visible, because you built your list in the reverse order.

You also use the `LayerManager` to present the contents of its list to the user on the display by invoking its `paint` method. When you invoke `paint`, the `LayerManager` renders each of its layers in order of descending index, implementing the Z-order promised by the interface. The `paint` method takes the `Graphics` instance to use when rendering the `Layer`s in the list, and the offset in the `Graphics` instance where drawing should take place. You use the `setViewWindow` method to set the clipping region that the `LayerManager` uses.

■ **Tip** The `LayerManager.paint` method is optimized; it won't render items that are completely outside the `Graphics` clipping region. Thus, if you create a custom `Layer` subclass and implement the `paint` method, you should be aware that the `paint` method is *only* invoked if there's something to paint.

Optimizing Visual Layers Using Tiling

Most games have at least one visual component that can consist of regularly repeating bitmaps, such as the background for a game level. For example, consider Figure 8-2, which shows an eight-by-eight-tile grid for a game board made up of three tiles.

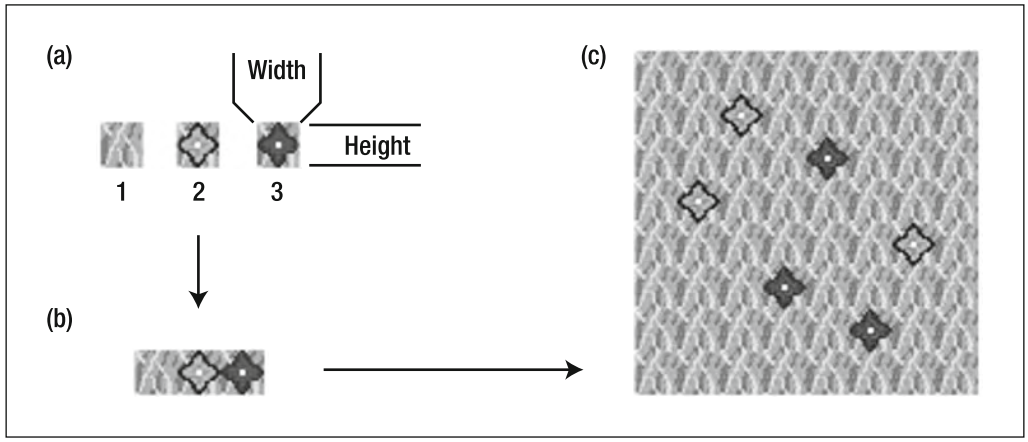


Figure 8-2. *The three tiles shown in (a) make up the tile image shown in (b), which is used repeatedly to create the tiled region in shown in (c).*

As you can see in Figure 8-2(a), each tile must have the same width and height. You can combine these tiles in a number of ways, as shown in the strip in Figure 8-2(b), so that each tile makes up one portion of an `Image` object. Finally, you can use multiple tiles in succession to create a larger image—such as a game background—as you see in Figure 8-2(c).

The `TiledLayer` class takes an image consisting of unique tiles, such as that shown in Figure 8-2(b), and a set of indexes you provide into the `Image` object, and fills the cells of an image with the tiles at each index. These indexes begin with the number `1`, indicating the tile at the upper-left corner of the image, and increment working left to right and top to bottom across the image you provide.

For example, you can compose Figure 8-2(c) using the `Image` in Figure 8-2(b) and the indexes into that image in Listing 8-4.

Listing 8-4. *The Array of Indexes into the Tiles That Form the Image*

```
1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 2, 1, 1, 1, 1, 1,
1, 1, 1, 1, 3, 1, 1, 1,
1, 2, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 2, 1,
1, 1, 1, 3, 1, 1, 1, 1,
1, 1, 1, 1, 1, 3, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1
```

Each cell of a `TiledLayer` image can either be a single tile in the source bitmap or be transparent, letting the layers behind the `TiledLayer` show through.

A TiledLayer's constructor takes everything necessary to create the TiledLayer image *except* the array of cell indexes, which you pass to its setCell method. Thus, to create a TiledLayer, you provide the following:

- The number of columns and rows in the tile image
- The image of tiles
- The width of a single tile in the image of tiles
- The height of a single tile in the image of tiles

For example, to create the TiledLayer used to draw the image in Figure 8-2(c), you would write something like the code shown in Listing 8-5.

Listing 8-5. *Creating the TiledLayer*

```
Image boardImage = Image.createImage(imageName);
private final int tileWidth = 16, tileHeight = 16;
private final int cellsWidth = 8, cellsHeight = 8;
board = new TiledLayer( cellsWidth, cellsHeight, boardImage, tileWidth, tileHeight);
```

Once you create the TiledLayer, you need to set the index of the image of each cell in the TiledLayer to the index of the tile in the tile bitmap. You do this using the setCell method, passing the coordinates of the cell you want to set and the index of the tile in the tile image, as shown in Listing 8-6.

Listing 8-6. *Using the setCell Method*

```
int[] map = {
    1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 2, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 3, 1, 1, 1,
    1, 2, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 2, 1,
    1, 1, 1, 3, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 3, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1
};

for( int i = 0; i < map.length; i++ ) {
    int x = i % cellsWidth;
    int y = i / cellsHeight;
    board.setCell(x, y, map[i]);
}
```

Tiles within a `TiledLayer` can be animated; that is, they can rotate through different tiles in succession—for example, when the illusion of moving water is desired. To do this, you need only denote each of the animated tiles as animated using the `createAnimatedTile` method and passing the index of the animated tile frame. The `createAnimatedTile` method returns a *negative* number, which you use when setting cell values using `setCell`. In turn, you can change all animated tiles of a given index to a different animated tile by invoking the `setAnimatedTile` method and passing the index of the original tiles to change and the index that they should be changed to.

Producing Animations

While the `TiledLayer` class is appropriate for creating large visible areas (perhaps with rudimentary animation within the visible area), the `Sprite` class is best used for small objects consisting of one or more bitmaps. You can animate `Sprite` objects by cycling through separate animation frames. Like the `TiledLayer`, these frames are given to a `Sprite` as a single `Image` object consisting of multiple frames, each with a constant width and height. Figure 8-3(a) shows the three frames of an animated butterfly, while Figure 8-3(b) shows one way you can composite those frames into a three-frame animation's source `Image`.

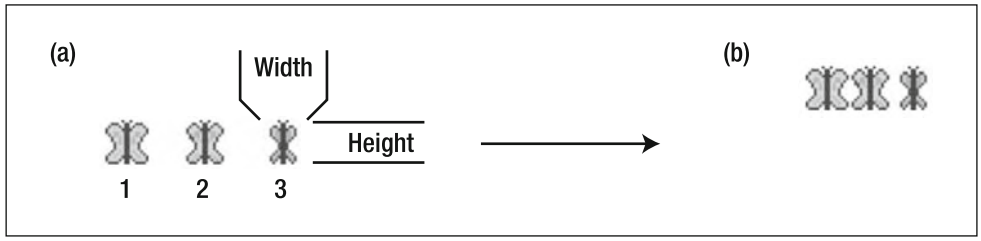


Figure 8-3. *As shown in (a), three frames make up the animation; in (b), these are combined into a single* `Image` *instance to pass to the* `Sprite` *class.*

Tip Transparency in the source `Image` is indicated in the usual way, such as by using the alpha channel of a PNG file.

Each frame is assigned a unique index; unlike tiles for a `TileLayer`, the `Sprite` class counts frames starting at `0`. Subsequent indexes are assigned from left to right and top to bottom. For example, to create a `Sprite` instance corresponding to the animation consisting of the frames in Figure 8-3(a), you might write the code shown in Listing 8-7.

Listing 8-7. *Creating a Sprite Instance*

```
private final int tileWidth = 16, tileHeight = 16;
Image image = Image.createImage(imageName);
Sprite butterfly = new Sprite(image, tileWidth, tileHeight);
```

Every `Sprite` has a frame sequence that defines an ordered list of frames to be displayed. By default, this is simply the order of the list of available frames. You can explicitly set this sequence using the method `setFrameSequence`, which takes an array of frame indexes, as shown in Listing 8-8.

Listing 8-8. *Setting the Frame Sequence*

```
private static final int[] flightSequence = {
    0, 1, 2, 2, 1, 0};
butterfly.setFrameSequence(flightSequence);
```

At any time, you can change which frame the `Sprite` will draw using any one of the methods `setFrame`, `prevFrame`, or `nextFrame`.

■ **Tip** Remember that `setFrame`, `prevFrame`, and `nextFrame` deal with indexes into the *frame sequence*, not the image of frames!

Each `Sprite` instance has a *reference pixel* that indicates the location from which the `Sprite` will draw its frame. By default, this is simply the upper left-hand corner of the frame, but you can change this position using the `defineReferencePixel`, passing the *x* and *y* coordinates of the new reference pixel. Reference pixels are especially handy when you consider that the `Sprite` class can apply various visual transformations to the frames that make up the `Sprite`. These transforms include rotations in multiples of 90° and mirroring around the vertical axis of each of these rotations. When the `Sprite` class applies a transformation, the `Sprite` is automatically repositioned so that the reference pixel appears stationary; for example, rotation occurs about the reference pixel. You can apply a transformation to a `Sprite` using the `setTransform` method, passing one of the constants shown in Table 8-2.

Table 8-2. *The Various Transformations You Can Apply to a Sprite's Frames*

Constant	Action
`Sprite.TRANS_NONE`	No translation
`Sprite.TRANS_ROT180`	Rotate 180° about reference pixel
`Sprite.TRANS_MIRROR`	Horizontal mirror about the reference pixel
`Sprite.TRANS_MIRROR_ROT180`	Rotate 180° about reference pixel, then horizontal mirror about reference pixel
`Sprite.TRANS_ROT90`	Rotate 90° clockwise about reference pixel
`Sprite.TRANS_MIRROR_ROT90`	Horizontal mirror about the reference pixel, then rotate 90° clockwise
`Sprite.TRANS_MIRROR_ROT270`	Horizontal mirror about the reference pixel, then rotate 90° counterclockwise
`Sprite.ROT270`	Rotate 90° counterclockwise about reference pixel

Finally, the `Sprite` class provides collision detection using the `collidesWith` method, checking to see if two `Sprite` instances (or a `Sprite` and an `Image`) collide. The `Sprite` class can run this collision test using either the rectangle of the frame image or the opaque pixels in the frame (which is a trifle slower); simply pass `true` if you want pixel-by-pixel collision detection. Using the `collidesWith` method can simplify your game, as you don't need to perform your own collision detection.

Putting the Mobile Game API to Work

Although writing a full game is beyond the scope of this chapter, it's instructive to see all of these pieces together in a full example. Figure 8-4 shows a simple game using the Mobile Game API that lets you move a cat about with the directional pad, chasing the moving butterflies. When the cat touches a butterfly, the handset's backlight flashes and the handset vibrates.

The application has two classes: one is responsible for implementing the MIDlet interface, and the other is a `GameCanvas` subclass that implements the game behavior itself. In addition, the MIDlet contains the artwork for the game, as shown in Listing 8-9.

Figure 8-4. *A simple game using the Mobile Game API*

Listing 8-9. *The MIDlet Contents*

```
src/
    com/
        apress/
            rischpater/
                SpriteCanvas.java
                SpriteSampleMIDlet.java
    res/
        butterfly-sprite.png
        cat.png
        ground-tiles.png
```

Let's look at each of these classes in more detail.

Implementing the Game MIDlet

The MIDlet itself is trivial: on launch, all it needs to do is create an instance of the custom canvas, set the new instance to be the current `Displayable`, and start the game thread. On termination, it should `null` out the canvas instance, set the current displayable to `null`, destroy itself, and notify the runtime that the MIDlet should be destroyed, as you see in Listing 8-10.

Listing 8-10. *The SpriteSampleMIDlet Class*

```
package com.apress.rischpater;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SpriteSampleMIDlet extends MIDlet {
    private SpriteCanvas canvas;

    public SpriteSampleMIDlet() {
    }

    private void initialize() {
        if (canvas==null) {
            try {
                canvas = new SpriteCanvas(getDisplay());
                getDisplay().setCurrent(canvas);
                canvas.start();
            }
        catch(Exception ex) {}
        } else {
            canvas.setPaused(false);
        }
    }

    public Display getDisplay() {
        return Display.getDisplay(this);
    }

    public void exitMIDlet() {
        destroyApp(true);
        notifyDestroyed();
    }
```

```
    public void startApp() {
        initialize();
    }

    public void pauseApp() {
        canvas.setPaused(true);
    }

    public void destroyApp(boolean unconditional) {
        canvas=null;
        getDisplay().setCurrent(null);
    }
}
```

This is straightforward code, especially if you already understand the life cycle of a MIDlet. The only other work this MIDlet class does is in `pauseApp`, where the MIDlet instructs the game canvas to pause game play by setting the game pause state to `true`. Game play is resumed when the MIDlet receives a new `startApp` invocation; if the game canvas already exists, the application will resume game play by setting the game pause state to `false`.

Implementing the Game Canvas

If the `SpriteSampleMIDlet` class is simple, the game canvas is only a little more complicated. This is due to the work in setting up the visible layers and the game loop. Listing 8-11 shows the `SpriteCanvas` class that extends the `GameCanvas` class and implements the game canvas and loop.

Listing 8-11. *The SpriteCanvas Class*

```
package com.apress.rischpater;

import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.io.IOException;
import java.util.Random;

public class SpriteCanvas
    extends GameCanvas
    implements Runnable {
```

```
private final int DELAYMS=75;
private boolean paused;

private Random random;
private Display display;

private LayerManager layers;
private TiledLayer board;
private Sprite[] butterfly;
private Sprite cat;

int[] map = {
    1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 2, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 3, 1, 1, 1,
    1, 2, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 2, 1,
    1, 1, 1, 3, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 3, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1 };
private static final int[] flightSequence = {
    0, 1, 2, 2, 1, 0 };
private final int tileWidth = 16, tileHeight = 16;
private final int boardWidth = 8, boardHeight = 8;

public SpriteCanvas(Display d)
    throws IOException {
    super(true);

    display = d;
    random = new Random();

    layers = new LayerManager();
    layers.setViewWindow(-(getWidth()-boardWidth*tileWidth)/2,
        -(getHeight()-boardHeight*tileHeight)/2,
         2* boardWidth*tileWidth, 2*boardHeight*tileHeight );
    createButterflies("/res/butterfly-sprite.png");
    createCat("/res/cat.png");
    createBoard("/res/ground-tiles.png");
}
```

```
    private void createBoard( String imageName )
        throws IOException {
        Image boardImage = Image.createImage(imageName);

        board = new TiledLayer( boardWidth, boardHeight,
            boardImage, tileWidth, tileHeight);

        for( int i = 0; i < map.length; i++ ) {
            int x = i % boardWidth;
            int y = i / boardHeight;
            board.setCell(x, y, map[i]);
        }
        layers.append(board);
    }

    private void createButterflies( String imageName )
        throws IOException {
        Image image = Image.createImage(imageName);
        int x, y, i;

        butterfly = new Sprite[2];

        for ( i=0; i<butterfly.length; i++ ) {
            butterfly[i] = new Sprite(image, tileWidth, tileHeight);
            x = random.nextInt(boardWidth)*tileWidth;
            y = random.nextInt(boardHeight)*tileHeight;
            butterfly[i].setPosition(x,y);
            butterfly[i].defineReferencePixel(0,0);
            butterfly[i].setTransform(Sprite.TRANS_NONE);
            butterfly[i].setFrameSequence(flightSequence);
            butterfly[i].setFrame(i % 3);
            layers.append(butterfly[i]);
        }
    }

    private void createCat( String imageName )
        throws IOException {
        Image image = Image.createImage(imageName);
        int x, y;
        int i;
```

```
        x = random.nextInt(boardWidth)*tileWidth;
        y = random.nextInt(boardHeight)*tileHeight;
        cat = new Sprite(image, tileWidth, tileHeight);
        cat.setPosition(x,y);
        cat.defineReferencePixel(0,0);
        cat.setTransform(Sprite.TRANS_NONE);
        cat.setFrame(0);
        layers.append(cat);
    }

    public void start() {
        Thread thread = new Thread(this);
        thread.start();
        paused=false;
    }

    private void detectCollisions() {
        int i;
        for (i=0; i<butterfly.length; i++) {
            if (cat.collidesWith(butterfly[i],true)) {
                display.flashBacklight(100);
                display.vibrate(100);
            }
        }
    }

    public void run() {
        Graphics g = getGraphics();
        while(true) {
            try {
                moveCat();
                moveButterflies();
                detectCollisions();

                layers.paint(g, 0, 0);
                flushGraphics();

                Thread.sleep(DELAYMS);
```

```
                synchronized(this) {
                    while (paused) {
                        wait();
                    }
                }
            }
            catch(InterruptedException ex) {}
        }
    }

    public void setPaused(boolean b)
    {
        synchronized(this) {
            paused = b;
            notify();
        }
    }

    private void moveCat() {
        int keyStates = getKeyStates();
        int x, y;
        x = cat.getX();
        y = cat.getY();
        if ((keyStates & LEFT_PRESSED) != 0) {
            x -= cat.getWidth()/4;
        }
        if ((keyStates & RIGHT_PRESSED) != 0) {
            x += cat.getWidth()/4;
        }
        if ((keyStates & UP_PRESSED) != 0) {
            y -= cat.getHeight()/4;
        }
        if ((keyStates & DOWN_PRESSED) != 0) {
            y += cat.getHeight()/4;
        }
        if ( x < 0 ) x = 0;
        if ( y < 0 ) y = 0;
        if ( x > board.getWidth() - cat.getWidth() )
            x = board.getWidth() - cat.getWidth();
        if ( y > board.getHeight() - cat.getHeight() )
            y = board.getHeight() - cat.getHeight();
        cat.setPosition(x,y);
    }
```

```
private void moveButterflies() {
    int dir;
    int i;
    int x, y, width, height;
    for ( i=0; i<butterfly.length; i++ ) {
        dir = random.nextInt(9) + 1;
        x = butterfly[i].getX();
        y = butterfly[i].getY();
        width = butterfly[i].getWidth();
        height = butterfly[i].getHeight();
        switch(dir)
        {
        /*  7 8 9
            4 5 6
            1 2 3   */
            case 1:
                x -= width/2;
                y += height/2;
                break;
            case 2:
                y += height/2;
                break;
            case 3:
                x += width/2;
                y += height/2;
                break;
            case 4:
                x -= width/2;
                break;
            case 5:
                break;
            case 6:
                x += width/2;
                break;
            case 7:
                x -= width/2;
                y -= height/2;
                break;
            case 8:
                y -= height/2;
                break;
```

```
                case 9:
                    x += width/2;
                    y -= height/2;
                    break;
            }
            // Clip coordinates
            if ( x < 0 ) x = 0;
            if ( y < 0 ) y = 0;
            if ( x > board.getWidth() - width )
                x = board.getWidth() - width;
            if ( y > board.getHeight() - height )
                y = board.getHeight() - height;
            butterfly[i].setPosition(x,y);
            butterfly[i].nextFrame();
        }
    }
}
```

Broadly speaking, the `SpriteCanvas` class methods can be broken up into three groups:

- *Game setup*: This is the responsibility of the constructor and the helper methods `createButterflies`, `createCat`, and `createBoard`.
- *Game play*: This is the responsibility of the game loop; it includes polling for keystrokes and moving nonplayer characters.
- *Handling pause and resume events*: In the event of a state change (say, due to an incoming call), the `SpriteCanvas` must pause game play. The MIDlet triggers this by invoking the `setPaused` method.

The constructor begins by invoking the `GameCanvas` constructor, indicating that this `GameCanvas` subclass will poll for keystrokes rather than receive keystroke events. Next, it caches aside the display its creator gives it, and it creates a new instance of `Random` used by the nonplayer characters. Finally, it creates and initializes the `LayerManager` (the skullduggery with the view window ensures that the game board will be centered in the display) and the various layers shown by the game. Note that the method creates the various layers in Z-order, with the butterflies closest to the user; this is because each creation routine adds the resulting `Layer` to the `LayerManager`.

The `createBoard` method creates an `Image` containing the tiles used to create the game background, and then it creates a `TiledLayer`, setting the cells in the `TiledLayer` to the tiles indicated by map. In a real game, this would be more complex, invoking a level loader that would load a specific game level from the game's resources, but the principle is essentially the same. Once the code initializes the `TiledLayer`, it adds it to the `LayerManager`.

The `createButterflies` method creates two butterflies by first loading the frame `Image` used in the butterflies' animation and then creating each butterfly `Sprite` in turn. For each butterfly `Sprite`, the code must select a random starting position for the butterfly, initialize the frame sequence (which shows how the butterfly's animation moves through the sequence of frames) and start the butterfly at a different frame than the previous butterfly. Once the method creates and initializes each butterfly `Sprite`, it adds it to the `LayerManager`. The `createCat` method is similar, except that the cat has no animation frames, so it omits the process of setting the frame sequence.

The game loop—started by the `start` method—must manage the movement of the cat and butterflies, detect any collisions between these characters, and redraw the screen. The `run` method accomplishes this work, together with the helper methods `moveCat`, `moveButterflies`, and `detectCollisions`. Finally, the game loop waits on the `paused` variable if `paused` is `true`, giving the MIDlet a way to pause the game.

The `moveCat` method begins by polling for keystrokes and testing for the directional keys. For each directional key pressed, the code incrementally shifts the cat's position; when all key combinations have been tested, the code checks to ensure the cat is still on the game board before setting the cat's position.

The `moveButterflies` method works similarly, with two differences. First, butterflies move randomly; second, `moveButterflies` must move all butterflies. It does so using a simple loop. Of course, this method and `moveCat` could be refactored to use a separate helper function, but I chose not to in order to keep the responsibilities of each method clear. In a more sophisticated game, this behavior might well be delegated to whole classes implementing the behavior of characters.

The code for `detectCollisions` delegates detecting collisions to the `Sprite cat` using its `collidesWith` method and passing each butterfly `Sprite` in turn. When `cat` detects a collision, the code uses the `Display` instance passed to the canvas's constructor to flash the backlight and vibrate the handset.

The game's main loop manages game pause and resume actions through the `setPaused` mutator, its associated `paused` variable, and Java's thread synchronization. When the member variable `paused` is `true`, the game loop in the `SpriteCanvas`'s `run` method waits, and the thread sleeps. When the MIDlet invokes `setPaused` again, it triggers a `notify` on the member variable `paused`, and the thread continues execution.

Wrapping Up

MIDP 2.0 contains the Mobile Game API in `javax.microedition.lcdui.game`, a robust set of classes you can use to implement your own games. This interface consists of five classes:

- `GameCanvas`: Lets you poll for keystrokes and structure your game around the notion of a game loop using Java threads
- `LayerManager`: Responsible for managing visible elements on the screen in a Z-order you define and draw
- `Layer`: Represents a single plane in the Z-order
- `TiledLayer`: An implementation of `Layer` you can use to create large bitmap images from small tiles
- `Sprite`: An implementation of `Layer` you can use to create animations

Using the Mobile Game API, you structure your game MIDlet around the notion of a game loop that runs in its own thread, managing events and updating game state. Your game loop can poll for keystrokes or respond to events in the traditional manner, and it draws to the screen as it needs to using the `GameCanvas`'s `Graphics` context. The `LayerManager` class simplifies drawing by managing an array of `Layer`s, which each can paint their own contents. The API defines two specific kinds of layers: the `TiledLayer`, which is suitable for drawing large regions of a game such as the game level's background, and the `Sprite`, which helps you organize the frames of an animation for a specific visible object.

Intermezzo

It may seem odd that Java ME encompasses the CDC, when few mobile devices to date actually implement it. However, when you consider the purpose of the CDC—to straddle the divide between low-cost, low-power computing devices and today's traditional portable or fixed workstation—it's clear that *something* needs to bridge that gap. Mobile devices and other consumer devices such as set-top boxes continue to gain processing power and memory, making it possible for ever-increasingly inexpensive devices to run applications suited for the CDC.

If you're sure you want to stick with MIDP programming—you've the makings of a dyed-in-the-wool mobile developer—feel free to skip Part 3 and move on to Part 4, where I show you more of the optional Java interfaces increasingly common on Java ME devices. On the other hand, if the possibility of other consumer devices excites you, read on! There's a lot more to Java ME than just the CLDC and its MIDP.

PART 3

CDC Development

Sun Microsystems has long seen Java as a platform for all computing devices, big or small. Arguably, Java's been successful for big jobs; technologies like Java Servlets power some of the world's big businesses. As you saw in Part 2, Java has been equally successful on some of the world's smallest devices—specifically, today's cheap, ubiquitous cell phones. Starting with PersonalJava, which evolved into the Java ME CDC, Sun is poised to repeat its success for mid-tier devices, including set-top boxes and other entertainment platforms. In this part, I show you how the CDC complements the CLDC, introducing the two key programming models for the CDC and its profiles: the Foundation, Personal Basis, and Personal Profiles. Don't assume that what you read applies only to entertainment devices or only to high-end mobile devices. Just as convergence has helped drive key features of Java SE into Java ME, in the years to come I expect what you see in these pages will be equally applicable to many Java-enabled non-traditional computing devices.

CHAPTER 9

Introducing Xlets and the Personal Basis Profile

The initial runtime environment for Java applets was the browser, but the advent of Java ME has changed that. As you saw in Chapter 4, the MIDlet is the MIDP's answer to an application runtime; in this chapter, you will see how the Xlet plays the same role for the Personal Basis Profile (PBP) atop the CDC. Unsurprisingly, the responsibilities an Xlet fulfils are essentially the same as those of a MIDlet, so if you're acquainted with the MIDlet, what you see here will be quite familiar.

In this chapter, I discuss the Xlet execution model. After reading this chapter, you will understand both the origin of the Xlet in the Java ME environment as well as the life cycle of an Xlet. You will be able to write your own Xlets, as well as perform basic inter-Xlet application communication using the interfaces available in the PBP.

Understanding the Xlet

The Xlet model comes to Java ME by way of Sun's foray into the television space for Java applications. Now part of the PBP atop the CDC, the Xlet model must meet some additional criteria above and beyond what a simple applet must; notably, it must permit multiple applications to share device resources, including the I/O devices. It also must ensure that no single application can bring down the Java virtual machine, as the virtual machine may be hosting other applications at the same time. Consequently, the Xlet must have a specific life cycle, letting the Xlet operate in active—running—and paused states, just like a MIDlet.

As with the MIDlet, you implement an Xlet by extending a specific interface—specifically, the `javax.microedition.xlet.Xlet` interface. As the runtime initializes the Xlet, it provides a *context*, which gives your Xlet some key methods to interact with the application runtime.

Looking at the Xlet Life Cycle

Figure 9-1 shows the states in which an Xlet may exist.

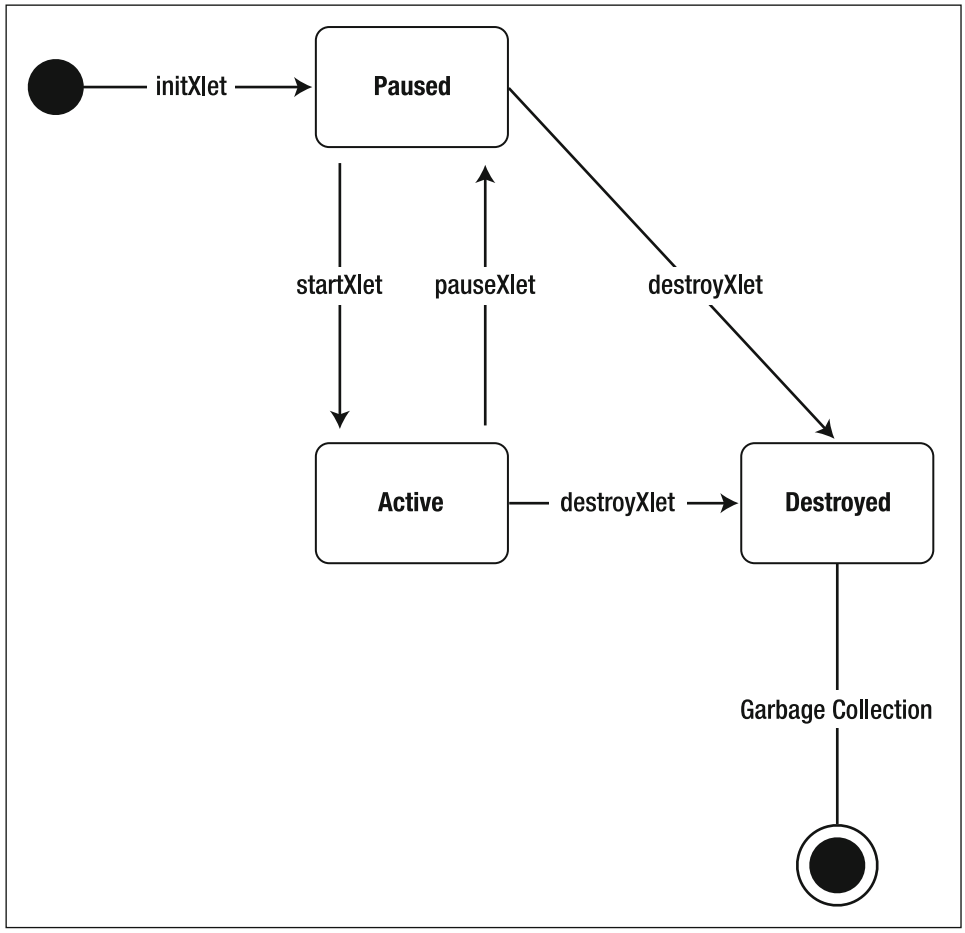


Figure 9-1. *The life cycle of an Xlet*

As you can see in Figure 9-1, the Xlet can be in one of three states:

- *Paused*: Once loaded, the Xlet application manager invokes the Xlet's `initXlet` method, which should initialize the Xlet. The Xlet is now in the *paused* state.

- *Active*: When the Xlet application manager is ready to give the Xlet control of the device's interface, it starts the Xlet using its `startXlet` method. The Xlet is now in the *active* state. At any time, the Xlet application manager can again pause the Xlet by invoking the Xlet's `pauseXlet` method, bringing the Xlet back to the paused state, from which it can again resume execution by invoking `startXlet` again.

- *Destroyed*: When the application manager wants to shut down the Xlet, it invokes the Xlet's `destroyXlet` method, which must terminate the Xlet's execution. The Xlet is now in the *destroyed* state, awaiting garbage collection by the runtime.

It's important to remember that an Xlet can move from the paused state straight to the destroyed state and that it may repeatedly reenter the active and paused states throughout its life. It's important to remember these two other things as well:

- *Don't call* `System.exit`: Yours may not be the only Xlet running, and you don't want to tear down the Java runtime. Use the `XletContext` object available to your Xlet (discussed later in this chapter) instead to signal when your application wants to close.

- *Free everything you can when paused and when exiting to help the garbage collector*: This is especially important for things such as remote communications interfaces, files, and the graphics context. The application should always be able to clean up after itself.

An interesting difference between the Xlet life cycle and the MIDlet life cycle is that an Xlet can throw an exception, `XletStateChangeException`, if it wants to ignore the requested state change. For example, an Xlet that fails to initialize can throw this exception to keep from starting, or an Xlet that is about to be destroyed can throw this exception to keep from being shut down by the application management system.

Extending the Xlet Interface

An Xlet must provide five methods:

- *A constructor*: This is generally empty.

- `initXlet`: This method takes an `XletContext` object and initializes the Xlet.

- `startXlet`: The runtime invokes this method as the Xlet enters the active state. This method must do whatever's necessary to start the Xlet's active execution.

- `pauseXlet`: This method must release unnecessary resources and indicate that the Xlet is in the paused state.

- `destroyXlet`: This method must release all resources as the Xlet enters the destroyed state and awaits garbage collection.

Listing 9-1 shows the simplest of Xlets.

Listing 9-1. *The Simplest of Xlets*

```
import javax.microedition.xlet.*;

public class SimplestXlet implements Xlet {
    public SimplestXlet () {
    }

    public void destroyXlet( boolean unconditional )
                        throws XletStateChangeException {
    }

    public void initXlet( XletContext context )
                        throws XletStateChangeException {
    }

    public void pauseXlet() {
    }

    public void startXlet() throws XletStateChangeException {
    }
}
```

This Xlet obviously doesn't do anything, but it illustrates an important point: it shows that that the Xlet interface implementation can also throw the `XletStateChangeException`. This exception indicates that a particular state transition has failed—for example, from loaded to initialized, or from running to paused, or the reverse. How the application management system handles these exceptions depends on the failed state transition and the implementation of the platform.

Using the Xlet Context

For each Xlet, the application manager associates a context, which is an instance of `javax.microedition.xlet.XletContext`. The runtime provides this manager so that the Xlet can interact with its environment. The `XletContext` interface provides the following methods:

- `getClassLoader`: Returns the base class loader of the Xlet
- `getContainer`: Returns the root container into which an Xlet should place its components
- `getXletProperty`: Returns a named property from the `XletContext`

- `notifyDestroyed`: Notifies the application manager that the Xlet has entered the destroyed state
- `notifyPaused`: Notifies the application manager that the Xlet does not want to be active and has entered the paused state
- `notifyActive`: Notifies the application manager that the Xlet wants to enter the active state

You typically use `getContainer` at your Xlet's initialization to obtain the container into which your other visible components should be placed. This is how the application manager shares the screen with multiple Xlets; each Xlet gets a separate containing component in which it can render its UI.

Like a MIDlet, an Xlet must signal the runtime when it wants to change state; in turn, the runtime honors the request, performing the state change. This is how an Xlet indicates that it wants to exit; instead of calling `System.exit`, it invokes `notifyDestroyed`. Similarly, an Xlet can indicate that it wants to enter the paused state and relinquish the UI to other Xlets by invoking `notifyPaused`, or it can request resumption to the active state by invoking `notifyActive`.

Writing a Simple Xlet

Writing an Xlet for the PBP is simple in the abstract: simply implement the Xlet interface, and in the process, extend a Java AWT container such as `java.awt.Component`. Your actual application resides inside this container, and you can interact with the application manager through the Xlet's context given to your Xlet at initialization time.

Looking at a Simple Xlet

In Listing 9-1, you saw the skeleton for an Xlet; Listing 9-2 shows an actual functioning Xlet.

Listing 9-2. *A Simple Xlet*

```
package com.apress.rischpater.HelloXlet;

import java.awt.*;
import javax.microedition.xlet.*;

public class HelloXlet extends Component implements Xlet {
    private XletContext context;
    private Container rootContainer;
```

```
    public HelloXlet() {
    }

    public void initXlet(final XletContext xletContext)
        throws XletStateChangeException {
        context = xletContext;
        if(rootContainer == null) {
            try {
                rootContainer = context.getContainer();
                rootContainer.add(this);
            } catch (UnavailableContainerException e) {
                System.out.println("Could not get our container!");
                throw new XletStateChangeException( "No container. "
                    + e.getMessage() );
            }
        }
    }

    public void startXlet() throws XletStateChangeException {
        rootContainer.setVisible(true);
    }

    public void paint(Graphics g) {
        g.drawString("Hello Xlet", 0, 100);
    }

    public void pauseXlet() {
        System.out.println("HelloXet.pauseXlet()");
    }

    public void destroyXlet(boolean b) throws XletStateChangeException {
        System.out.println("HelloXet.destroyXlet() - goodbye");
    }
}
```

Figure 9-2 shows the Xlet's output when run in the Java ME emulator.

Figure 9-2. *Hello Xlet*

The Xlet itself is simple: implementing the `Xlet` contract, it extends `Component`, which is a lightweight AWT class that knows how to draw when its `paint` method is invoked.

On initialization, the Xlet must do several things. First, it caches aside the context the application management system provides to use when it wants to exit. Next, it obtains the root container provided by the context and caches it aside; this container defines where on the screen the Xlet is permitted to draw. Finally, it adds itself to the context-provided root container.

By default, the Xlet's UI is not visible. Thus, when the Xlet transitions to the active state as the application manager invokes `startXlet`, the application must make its root container visible; this causes a repaint that shows the application interface, handled by `paint`, which performs a simple `Graphics` operation to paint the message "Hello Xlet" on the display.

The Xlet shouldn't erase itself on pause; it may still be a visible part of the screen, just not focused. Consequently, pausing this Xlet is trivial, because it doesn't do anything; the `pauseXlet` method simply logs to the console the fact that the application manager triggered a transition to the paused state.

Similarly, destroying this Xlet is also trivial; the AMS handles removing the frame from the root container, which triggers the destruction of the UI, so all destruction does is log the fact that the Xlet was destroyed. (There's no need for the Xlet to hide itself, either, as the application manager does this for you.)

Understanding Xlet Dependencies

It's worth observing that this Xlet, while simple, actually has two key dependencies beyond the CDC, and these dependencies are challenges you may face in designing your own Xlet. First, and most obvious, you must remember that the Xlet isn't a CDC class; it's a PBP class. What that means to you is that just because a device supports the Java ME CDC, that doesn't necessarily mean that it provides the Xlet runtime, unless it implements the PBP. Fortunately, the vast majority of today's CDC devices do include at least the PBP, so in practice this isn't likely to be a problem.

Second, the core CDC provides the Java class hierarchy for connected Java ME devices, but it explicitly delegates the responsibility of determining how applications run and what window toolkit is available to the various profiles atop the CDC. In addition to providing the Xlet model, the PBP also provides a small subset of the Java AWT: components and containers. Specifically, the PBP supports those AWT components that have no peers in the operating system, letting application and framework developers implement their own window toolkit. In fact, several platform vendors have done just this, and if you're working with Java ME on some devices, you may not be using a standard window toolkit like the AWT at all, but instead a different component framework provided by a third-party vendor.

Fortunately, there are other options for many more capable devices. Several JSRs detail specific user-interface packages for CDC-enabled devices, including Scalable Vector Graphics (SVG), available in a mobile profile format suitable for CDC-enabled devices. JSR 226 defines an interface that supports SVG via the `javax.microedition.m2g` package, and JSR 287 defines the second iteration of that interface with support for events and the SVG Document Object Model (DOM). In addition, on some devices, three-dimensional graphics may be available via the Mobile 3D Graphics API that JSR 184 defines, or the OpenGL ES Common profile interfaces defined in JSR 239.

Significantly more useful for many Java developers is the AGUI that JSR 209 defines. This defines a reasonable subset of both Java AWT and Swing user interfaces that facilitates porting Swing applications to Java ME devices. I discuss the AGUI in more detail in Chapter 10.

Of course, all of these window toolkit options have underlying dependencies on the core configuration and profiles that a particular device supports; Figure 9-3 shows typical stacks of configurations, profiles, and GUI profiles defined by the JSR process.

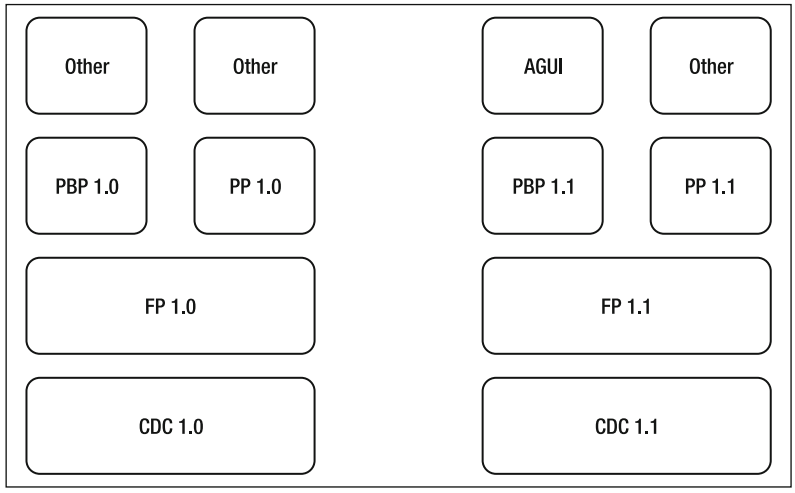


Figure 9-3. *The relationship between the CDC, the profiles atop the CDC, and some of the more commonly available GUI toolkits available for the CDC*

The figure shows two stacks: one for CDC 1.0, and one for CDC 1.1. Virtually any platform running a window toolkit supports the Foundation Profile (FP) atop the CDC; as you learned in the "Introducing the Foundation Profile" section in Chapter 1, the FP provides network and I/O support, but not application support, which is the responsibility of either the PBP or the Personal Profile (PP), which includes the full AWT as well as the Xlet model. Potentially atop either the PBP or the PP are other window toolkits, either proprietary or based on JSRs such as the ones for SVG, Mobile 3D, or OpenGL ES.

On CDC 1.1, the stack looks much the same, except that both PBP and PP have minor tweaks and a new version number, and the AGUI providing most of AWT and Swing, in addition to the possibility of other toolkits such as SVG, Mobile 3D Graphics, or OpenGL ES.

As an example of how a consumer electronics device based on Java ME might implement a window toolkit stack, consider the Blu-ray Disc Java (BD-J) stack's approach to the UI, shown in Figure 9-4. Here, the Home Audio Visual Interoperability (HAVi) Group window toolkit, defined by the `havi.ui` package, sits directly atop the PBP, and in fact it implements a rich collection of UI widgets using only the lightweight components such as `java.awt.Component` provided by the PBP.

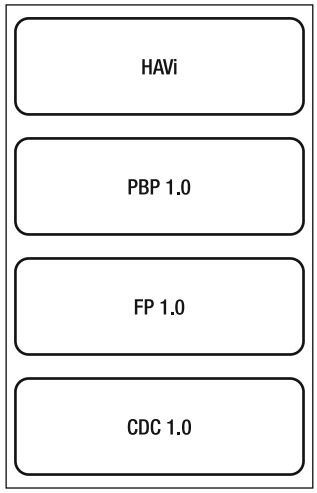


Figure 9-4. *The BD-J software stack*

Listing 9-3 shows the same Hello Xlet example, this time written using the HAVi classes provided as part of the BD-J stack.

■ **Note** Access to the BD-J stack is limited to developers actually developing Blu-ray content; typically, you need to obtain a license to a Blu-ray authoring environment that supports BD-J development.

Listing 9-3. *Hello Xlet Written Using the HAVi Window Toolkit Classes*

```
package com.apress.rischpater.HelloHaviXlet;

import java.awt.*;
import org.havi.ui.*;
import javax.microedition.xlet.*;

public class HelloHaviXlet extends Component implements Xlet {
    private HScene scene;

    public void initXlet(XletContext context)
        throws XletStateChangeException {
        scene=HSceneFactory.getInstance().getBestScene(new HSceneTemplate());
        scene.add(this);
    }
```

```
    public void startXlet() throws XletStateChangeException {
        scene.setVisible(true);
    }

    public void pauseXlet() {
    }

    public void destroyXlet(boolean b) throws XletStateChangeException {
        scene.dispose();
    }

    public void paint(Graphics g) {
        g.drawString("Some text", 0, 100);
    }
}
```

Fundamentally, the code in Listing 9-3 looks very much the same as the code in Listing 9-2. The HAVi toolkit defines the notion of *scenes*, which you can think of as being analogous to containers; the `initXlet` method creates a new scene and positions it on the display. Otherwise, the code is the same as the previous example, except that you must explicitly dispose of a scene when you're through with it; this Xlet does this as part of the `destroyXlet` method.

Clearly, then, when setting out to develop Xlet applications, you should have answers to the following questions before you begin your design:

- What version of the CDC is supported?
- What version of the FP is supported?
- What version of the PBP—or the PP—is supported?
- Which additional window toolkits, if any, are available?

In Chapter 10, after a brief discussion of applet development for the PP's support for the Applet model, I discuss programming for the AWT and AGUI in more depth.

Developing Lightweight User Interfaces Using the PBP

In the Java SE AWT, the entire AWT hierarchy descends from the `Component` class and its subclass `Container`, and the class hierarchy acts as a proxy for a native UI component; for example, the `java.awt.Checkbox` class is just a proxy for a corresponding peer component that wraps the behavior of the native platform's window toolkit. On some PBP-enabled

hosts, there may be no native window toolkit other than that provided by the Java runtime; on others, the amount of memory consumed by the peer hierarchy would be too great for the platform to support. To get around this, as you have already learned, the PBP does not contain a full user-interface hierarchy for your application development. In fact, the PBP explicitly *excludes* heavyweight components such as text labels, text input boxes, buttons, lists, and menus. Instead, it requires the following four AWT top-level component classes:

- `java.awt.Component`: The base class for all user-interface components
- `java.awt.Container`: The base class for AWT components that contain other components
- `java.awt.Frame`: A top-level window with a title bar and a border
- `java.awt.Window`: The base class for top-level windows

Believe it or not, these four core component classes are enough to implement an entire window toolkit, because the `Component` and `Container` classes can represent components including nested components, and the `Frame` and `Window` classes permit you to define top-level windows that contain those components. Of course, the PBP-required AWT subset includes a host of additional classes that permit you to implement your user interface, including `java.awt.Graphics`, `java.awt.Image`, and `java.awt.AWTEvent`.

This leaves you with just two choices for building an application that uses a GUI, as you saw in the previous section: you can either use one in a package provided by someone else, or you can roll your own. The Java community has defined several such window toolkits, including more of the standard Java classes for the UI (AWT and/or Swing, depending on the profile and packages on the target), and there are others such as the HAVi hierarchy for BD-J. In addition, creating your own hierarchy may well be feasible for your application, especially if your user interface has simple requirements.

Implementing Your Own Components for a Window Toolkit

In the Java user-interface paradigm, a *component* is any object that has a graphical representation that can be displayed on the screen and interact with the user. A *container* is a component that can also contain other components. When implementing your own GUI, most of the work you need to do involves creating your own primitive components, such as buttons. This involves implementing subclasses of `java.awt.Component`.

The key to implementing your own toolkit is handling the drawing for each of your toolkit's components. Typically, this happens for one of two reasons: system-triggered painting occurs, such as when the component is made visible on the screen or is resized, or application-triggered painting occurs, when the component decides it needs to repaint itself because its contents have changed. The AWT contract—which you must

fulfill by implementing the `Component` interface—requires that drawing occur via callbacks to the component, specifically through its `paint` method. Your `paint` method is passed a `Graphics` object, which has all of the primitives you require to perform two-dimensional painting on a canvas.

It's important that you refrain from caching the `Graphics` object you receive when the system invokes your `paint` method, or from painting in methods other than your `paint` method. Other methods may be invoked at inappropriate times for your component to draw, such as when it's not visible or when the `Graphics` object isn't in a state to perform drawing. When the framework invokes your `paint` method, the `Graphics` object is preconfigured with the appropriate state for drawing your component, including the following graphics rendering options:

- The `Graphics` object's color is set to the component's `foreground` property.
- The `Graphics` object's font is set to the component's `font` property.
- The `Graphics` object's translation is set such that the origin (0,0) represents the upper-left corner of the component.
- The `Graphics` object's clip rectangle is set to the area of the component that needs to be redrawn.

Of course, you're free to reconfigure the `Graphics` object passed to your `paint` method as necessary.

To signal to the component that it should redraw, the application—or the component itself—should invoke one of the component's `repaint` methods. When invoking `repaint`, whenever possible you should include the bounds that must be repainted, so that the component can limit the redrawing to the region that's necessary. You can also pass a time interval in milliseconds, indicating that the component should repaint itself before the specified number of milliseconds elapses. This form of `repaint` is especially useful when performing multiple repaints, because it lets you queue up all of the drawing work at once.

Note that although it looks similar, the `update` method provided by a component isn't the same as `repaint`. Only the AWT event system invokes `update` in response to an application-triggered redraw request; system repaints do not trigger a call to `update`. This lets you hook either system-level redraw requests (by overriding `update`) or application-level redraw requests (by overriding `repaint`). However, odds are that you will want the default component behavior for both `repaint` *and* `update`.

`Container` subclasses, which can contain more than one component, are themselves `Component` subclasses and can also perform their own drawing by overriding the `paint` method. The default implementation of the `Container`'s `paint` method is to invoke `paint` on any of its visible children that intersect the rectangle to be painted, so if you're implementing a `Container` subclass that knows how to paint, it's imperative that your container's `paint` method invoke the superclass's `paint` method, as shown in Listing 9-4.

Listing 9-4. *Invoking the Superclass's paint Method*

```
public class PaintingContainer extends Container {
    public void paint(Graphics g) {
        // paint my contents first…
        super.paint(g);
    }
}
```

A full discussion of the `Graphics` class and its methods is beyond the scope of this chapter, but it's important to remember that a `Graphics` object both stores the context of a particular set of drawing primitives (font, color, clip region, and so on) and provides methods to draw graphic primitives, including rectangles, arcs, lines, polygons, characters, and images. The last primitive—the ability to draw images—may be key for your window toolkit components, because you can have an artist draw images for various parts of a component (such as a button's boundary, background, and so forth), and then your `Component` subclass can simply composite those images with the `Graphics` object in its `paint` method with a minimum of primitive graphics to provide a specific look and feel.

Writing a Simple, Lightweight Component

Implementing a rich windowing toolkit *can* be a great deal of work, but it doesn't have to be, especially if your application only has a few user-interface primitives. Listing 9-5 demonstrates a simple component: a round button bearing a text label.

Listing 9-5. *The RoundButton Lightweight Component*

```
import java.awt.*;
import java.lang.*;
import java.util.*;
import java.awt.event.*;

public class RoundButton extends Component {
    protected String label;
    protected boolean pressed = false;
    private Image offscreen;
    private static int PREFERRED_SIZE = 100;
    private static int LABEL_PAD = 40;

    public RoundButton(String l) {
        label = l;
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    }
```

```
public void invalidate() {
    super.invalidate();
    offscreen = null;
}

public void update(Graphics g) {
    paint(g);
}

public void paint(Graphics g) {
    int s = Math.min(getSize().width - 1, getSize().height - 1);
    // Double-buffer the drawing
    if(offscreen == null) {
       offscreen = createImage(getSize().width, getSize().height);
    }
    Graphics og = offscreen.getGraphics();
    og.setClip(0,0,getSize().width,getSize().height);

    // Draw the background, indicating press state.
    if(pressed) {
        og.setColor(getBackground().darker().darker());
    } else {
        og.setColor(getBackground());
    }
    og.fillArc(0, 0, s, s, 0, 360);

    // draw the perimeter of the button
    og.setColor(getBackground().darker().darker().darker());
    og.drawArc(0, 0, s, s, 0, 360);

    // draw the label centered in the button
    Font f = getFont();
    if(f != null) {
        FontMetrics fm = getFontMetrics(getFont());
        og.setColor(getForeground());
        og.drawString(label,
                      s/2 - fm.stringWidth(label)/2,
                      s/2 + fm.getMaxDescent());
    }
    g.drawImage(offscreen,0,0,null);
    og.dispose();
}
```

```
    public Dimension getPreferredSize() {
        Font f = getFont();
        if(f != null) {
            FontMetrics fm = getFontMetrics(getFont());
            int max = Math.max(fm.stringWidth(label) + LABEL_PAD,
                               fm.getHeight() + LABEL_PAD);
            return new Dimension(max, max);
        } else {
            return new Dimension(PREFERRED_SIZE, PREFERRED_SIZE);
        }
    }

    public Dimension getMinimumSize() {
        return new Dimension(PREFERRED_SIZE, PREFERRED_SIZE);
    }

    public void processMouseEvent(MouseEvent e) {
        switch(e.getID()) {
            case MouseEvent.MOUSE_PRESSED:
                pressed = true;
                repaint();
                break;
            case MouseEvent.MOUSE_RELEASED:
                if(pressed == true) {
                    pressed = false;
                    repaint();
                }
                break;
            case MouseEvent.MOUSE_ENTERED:
                break;
            case MouseEvent.MOUSE_EXITED:
                if(pressed == true) {
                    pressed = false;
                    repaint();
                }
                break;
        }
        super.processMouseEvent(e);
    }
}
```

The `RoundButton` shows all of the principles I discuss in the previous section, and it uses double buffering to an offscreen `Image` instance to prevent the possibility of flickering when the component repaints. It's not a complex component, but it is a little long, so let's take it field by field and method by method.

The `RoundButton` has a `label` field that contains the string labeling the button for the user. It must also track its state, whether it's pressed or not, which it does using the `protected pressed` field. It maintains an `Image` for offscreen drawing, aptly named `offscreen`, and two private constant variables: the preferred size of the button and the default padding for the button label.

Tip You may be wondering why I chose `protected` for the access protection level of these fields. I'm assuming that this class will be part of a more sophisticated GUI framework, and thus might have subclasses that customize the behavior of this button and need access to its fields.

The `RoundButton` constructor simply caches aside the label you provide and enables mouse events for the component. The `RoundButton` must override the `invalidate` method as part of the double buffering. The windowing environment calls `invalidate` as part of the component's sizing and layout, which in turn requires the `RoundButton` to allocate a new offscreen bitmap for drawing. For speed, the `invalidate` method only invalidates the current offscreen bitmap; it leaves allocating the new offscreen bitmap to the `paint` method.

The `update` method—invoked by the windowing toolkit hierarchy as a result of a system redraw—must, by definition, clear the component's background and completely redraw the component. Because `paint` draws the entire component to an offscreen bitmap and then copies that bitmap to the screen, `update` can simply call `paint` directly.

The `paint` method does the real work for the `RoundButton`. It begins by creating the offscreen bitmap if one is required, and then it obtains a `Graphics` instance for the offscreen bitmap. It then draws the button on the offscreen bitmap from back to front, with the following operations:

1. It computes the button background, which is slightly darker if the button is being pressed.

2. It draws a filled circle for the button using the color it computed in the previous step.

3. Once it has drawn the background, it draws the border using a shade darker than the button's background.

4. With the background and border drawn, it draws the button's label, which is centered in the region defined by the button.

5. Now that it has completely updated the offscreen bitmap, it draws the image corresponding to the offscreen bitmap using the current graphics context.

6. It disposes of the offscreen bitmap's graphics context.

The `getPreferredSize` and `getMinimumSize` methods provide clues regarding the component's desired size to any layout managers. These methods take into account the size of the label for the button.

The `RoundButton` needs to only handle mouse events, which it does using the `processMouseEvent` method; when you press the mouse and the mouse cursor is within the bounds of the button, it should visibly show the press by darkening its background; this darkening must disappear when you release the mouse button within the bounds of the `RoundButton`, or when the mouse cursor exits the `RoundButton`'s bounds *and* you've previously pressed the `RoundButton`. The `switch` statement within the `processMouseEvent` method handles the various cases for this logic. `processMouseEvent` also invokes `repaint` to force the `RoundButton` to redraw to visibly show its changed state.

Understanding Window Toolkit Limitations of the PBP

You've just seen a key limitation of the window toolkit provided by the PBP: the proscription of heavyweight components. However, there are other limitations that you should be aware of:

- The number, size, and location of `java.awt.Frame` instances—which correspond to native windows—may be severely limited.

- The decoration, title, and resizability of `java.awt.Frame` instances are under the control of the native window manager and are guidelines, not requirements. The native window toolkit is free to disregard requests from your application for decoration, including title, size, and placement.

- The platform may or may not have a mouse, and thus, it may not generate mouse events.

- The platform may or may not have a full keyboard, and thus, it may not generate keyboard events.

- Alpha compositing may not support full compositing of a source over a destination.

- The `BasicStroke` attributes may be ignored when specifying a graphics stroke.

- You may not be able to set the cursor bitmap on a component-specific basis.

In practice, the upshot of these restrictions is that many PBP implementations support only a single top-level window, so you can't construct applications with more than one window. To determine whether or not the PBP implementation you're working with has these limitations, you can query a system property for each limitation. Table 9-1 shows the system properties and their values.

Table 9-1. *PBP System Properties and Their Values*

Property	Value
`java.awt.AlphaComposite.SRC_OVER.isRestricted`	`true` if and only if `AlphaComposite.SRC_OVER` is restricted
`java.awt.Graphics2D.setStroke.BasicStroke.isRestricted`	`true` if and only if the use of `BasicStroke` is restricted in `Graphics2D.setStroke`
`java.awt.event.MouseEvent.isRestricted`	`true` if and only if `MouseEvent` is restricted
`java.awt.event.MouseEvent.supportLevel`	Level of support for `MouseEvent`, if restricted; undefined otherwise[1]
`java.awt.event.KeyEvent.isRestricted`	`true` if and only if `KeyEvent` is restricted
`java.awt.event.KeyEvent.supportMask`[2]	Mask describing `KeyEvent` support, if restricted; undefined otherwise
`java.awt.Component.setCursor.isRestricted`	`true` if the cursor image cannot be changed for any `Component`; optionally supported by platforms
`java.awt.Frame.setLocation.isRestricted`	`true` if `Frame` location is limited to a single value; optionally supported by platforms
`java.awt.Frame.setResizable.isRestricted`	`true` if `Frame` resizability may not be changed; optionally supported by platforms
`java.awt.Frame.setSize.isRestricted`	`true` if `Frame` size is limited to a single value; optionally supported by platforms
`java.awt.Frame.setTitle.isRestricted`	`true` if `Frame` titles may not be changed; optionally supported by platforms
`java.awt.Frame.setUndecorated.isRestricted`	`true` if `Frame` decorations may not be changed; optionally supported by platforms

[1] *The* `java.awt.event.MouseEvent.supportLevel` *property is* `0` *if the platform generates no mouse events,* `1` *if the platform provides all events but pointer movement events, and* `2` *if it provides all events that the AWT defines.*

[2] *The* `java.awt.event.KeyEvent.supportMask` *is a bit mask where* `1` *indicates support for* `VK_LEFT` *and* `VK_RIGHT` *keys,* `2` *indicates support for* `VK_UP` *and* `VK_DOWN` *keys,* `4` *indicates support for* `VK_0` *through* `VK_9` *(a numeric keypad), and* `8` *indicates support for* `VK_A` *through* `VK_Z` *with* `VK_SPACE` *and* `VK_BACK_SPACE` *(an alphanumeric keypad).*

Obtaining Xlet Properties and Resources

As you saw in the section "Using the Xlet Context" in this chapter, the `XletContext` provided by the application runtime to your Xlet includes the `getXletProperty` method for obtaining runtime properties. The only property presently defined by the Java ME standard itself is the `XletContext.ARGS` property, which permits your application to collect any command-line launch arguments passed when the user launched your application, letting the application manager pass options to your Xlet.

The Xlet can also interrogate the system for properties using the static class method `System.getProperty`, which typically returns system-level properties proprietary to the platform, rather than runtime properties defined by the PBP or PP implementation on which your application is running. For Java ME consumer devices aimed at the audio-visual market, for example, these include information about the file system, the broadcast profile, the availability of Internet access, and the version number of additional packages such as the UI widget set (for example, the HAVi package).

Xlets are typically packaged as JAR files, so you can include additional resources with your application as part of the associated JAR file. To access other resources in your Xlet's JAR file, your implementation must use the Xlet context's class loader, just as it would for a stand-alone application or applet. For example, you might write the code shown in Listing 9-6.

Listing 9-6. *Accessing an Image from an Xlet JAR File*

```
ClassLoader cl = context.getClassLoader();
Icon icon  = new ImageIcon(cl.getResource("images/icon.png"));
```

Using the class loader provided by the `XletContext`, you can get application and system resources by name, and you can obtain either a URL to the resource or an `InputStream` to the resource using one of the following methods:

- `getResource`: Returns a URL to the named resource
- `getResourceAsStream`: Returns an `InputStream` to the named resource
- `getSystemResource`: Returns a URL to the named system resource, which is a platform-specific attribute of the target hardware
- `getSystemResourceAsStream`: Returns an `InputStream` to the named system resource, which is a platform-specific attribute of the target hardware

Communicating with Other Xlets

A key difference between the MIDlet execution model provided by the MIDP and that of the Xlet execution model provided by the PBP is that the Xlet execution model requires that Xlets support rudimentary Inter-Xlet Communication (IXC). Data sharing for Xlets is increasingly important, as the platforms that can run Xlets typically have access to a great deal of data (through either broadcast or wide-area networks such as the Internet) but limited storage for applications and data. As a result, data sharing between applications becomes increasingly important, because it is impractical for each application to contain the code necessary to access and decode a network source for data and cache its own copy of the data it needs in order to function.

The usual means for applications to share data in the Java environment is through the Java RMI stack, which permits both interapplication and interdevice communication. In fact, a robust subset of the RMI stack *can* be made available for Java ME; this is discussed in JSR 66, and I discuss it in more detail in Chapter 11. This support for RMI is key for some set-top boxes and other platforms with a strong reliance on middleware, because RMI enables servers and clients to exchange data in a Java-native, object-oriented way, simplifying the design and deployment of multitier systems.

Not all PBP devices need the entire RMI stack, however, so the PBP defines only a subset of RMI suitable for letting Xlets share data. Not surprisingly, the PBP mechanisms closely mirror the RMI architecture, leveraging some classes and the same basic process for defining the communication process.

IXC requires that data be represented by shared Java objects that encapsulate the data you want to share. These objects must be *remotable*—that is, they must implement the `java.rmi.Remote` interface. Xlets share these remotable objects through the IXC registry, which is a singleton shared by all Xlets running on the local host available from the `XletContext`. Xlets can register the remotable object using the IXC registry, or they can obtain access to remoted objects through the registry. Figure 9-5 shows a schematic of the relationship between two different Xlets and an object they share.

As the diagram suggests, there's a subtle difference between the object shared through the registry by the first Xlet, and what the second Xlet actually consumes. Due to the way the Java class loader operates, Xlets cannot simply share objects; instead, the IXC registry creates and returns a stub object that acts as a proxy to the shared object. This is similar to the RMI model, although in the full RMI implementation, either you must generate the stub using the RMI compiler (if you're using a version of Java before Java 5) or the RMI runtime can do this automatically (if you're using a version of Java after version 5).

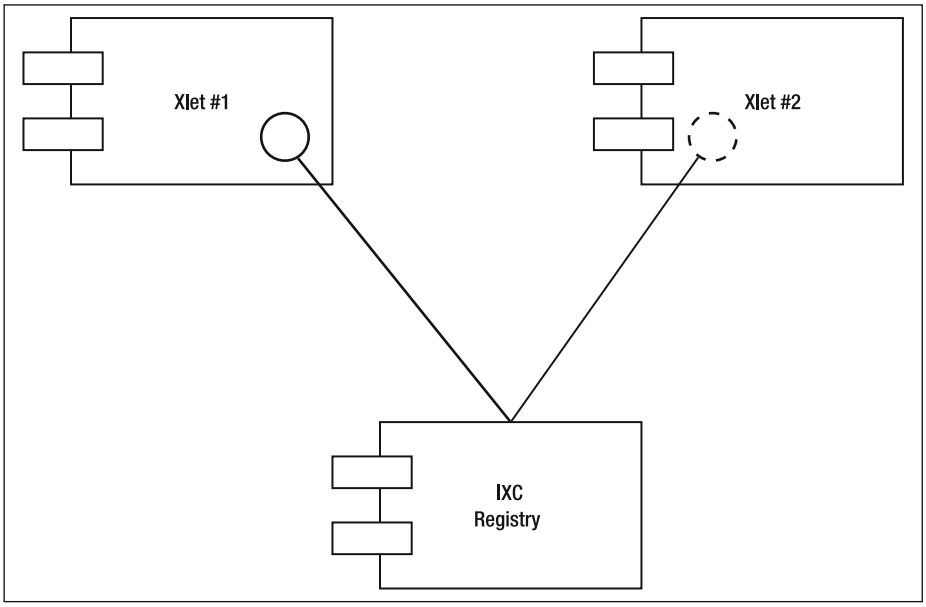


Figure 9-5. *The relationship between two Xlets and an object they share*

Implementing a Shared Object

Consider an Xlet with explicit access to a network that wants to share data with other applications. You must wrap this data in a remotable class, as you see in Listing 9-7.

Listing 9-7. *A Data Wrapper Class Exported Using IXC*

```
import java.rmi.*;

public class Location
    implements java.rmi.Remote {

    private String location;
    private String forecast;

    public Location() {
    }

    public void setLocation(String l)
        throws RemoteException {
        location = l;
    }
```

```
    public void setForecast(String f)
        throws RemoteException {
        forecast = f;
    }

    public String getLocation()
        throws RemoteException {
        if (location != null) {
            return location;
        }
        else {
            return "";
        }
    }

    public String getForecast()
        throws RemoteException {
        if (forecast != null) {
            return forecast;
        }
        else {
            return "";
        }
    }
}
```

Pretty simple stuff, except for two key differences between this class and a generic container class for data: the declaration imports `java.rmi`, and each of the accessors and mutators can throw `java.rmi.RemoteException`. When implementing a remotable class, all of its methods may throw `java.rmi.RemoteException`, because the class implementation may actually be a remote stub provided by the IXC registry and may be unable to communicate with the actual instance of the class.

As I previously stated, the entire `java.rmi` class hierarchy is *not* available to Xlets running atop the PBP. Instead, the PBP requires only the following RMI classes:

- `java.rmi.Remote`: Used to indicate a remotable class
- `java.rmi.AccessException`: Thrown to indicate that the caller of a remote object does not have permission to perform the requested action
- `java.rmi.AlreadyBoundException`: Thrown if you attempt to bind an object to the registry to a name that has already been bound to another object

- `java.rmi.NotBoundException`: Thrown if you attempt to look up or unbind an object that has no associated binding
- `java.rmi.RemoteException`: The common superclass for communications-related exceptions that may occur during remote method invocation
- `java.rmi.UnexpectedException`: Thrown if the client of a remote method call receives an exception that is not among the exception types declared in the `throws` clause of the method in the remote interface
- `java.rmi.registry.Registry`: A remote interface to a remote object registry

It's important to remember that the IXC mechanism lets you invoke the methods of objects in another Xlet's context, rather than share data in its simplest sense. In other words, you can't obtain or mutate attributes of an object through IXC and objects that implement `java.rmi.Remote`; instead, you must make attributes available through methods. Given that this is good object-oriented practice anyway, it shouldn't be an issue, but it does mean defining appropriate accessor and mutator methods for any field in your class that you want to be available to a remote Xlet. That's why `Location` defines `setLocation`, `setForecast`, `getLocation`, and `getForecast` as mutators and accessors to private fields, rather than simply making those fields public.

Sharing an Object for Other Xlets to Find

Once you define your remotable object(s), you still need to make them available to other Xlets. Doing this is a two-step process: first, you create the objects your Xlet will share, and then you add them to the system-wide Xlet registry. The registry, which is an instance of `javax.microedition.xlet.ixc.IxcRegistry`, keeps a name-object mapping of the objects registered by all Xlets running on the machine.

■ **Caution** Don't confuse the `IxcRegistry` with the `java.rmi.Naming` or `java.rmi.registry.LocateRegistry` registries; even if RMI is available on the machine, the `IxcRegistry` is a separate registry of remotable objects.

You can't obtain an instance of `IxcRegistry` directly; instead, you obtain one from the `XletContext`. Under the hood, the PBP implementation maintains a strict one-to-one correspondence between `IxcRegistry` instances and `XletContexts`, sharing data through a system-wide database of IXC exported objects. Using the `IxcRegistry`, you can do the following:

- Obtain an `IxcRegistry` instance for a specific context by invoking its static `getRegistry` method.
- Register an object implementing `java.rmi.Remote` to a unique name using the `bind` method.
- Return an array of `String`s naming each bound object in the registry using `list`.
- Call `lookup` to retrieve a specific `Remote` object.
- Unregister a specific object using `unbind`, or unregister all objects registered in the current `XletContext` using `unbindAll`.
- Register a new object implementing `java.rmi.Remote` to an already registered name using the `rebind` method.

Listing 9-8 shows how a prototypical Xlet might offer a remote `Location` instance to other Xlets.

Listing 9-8. *An Xlet Producing an Object for IXC Consumption*

```
import java.rmi.*;
import javax.microedition.xlet.*;
import javax.microedition.xlet.ixc.*;

public class XletLocationProducer implements Xlet {
    private XletContext context;
    private Location location;
    private static final String NAME =
        "XletLocationProducer.Location";

    public XletLocationProducer () {
    }

    public void initXlet( XletContext c )
        throws XletStateChangeException {
        location = new Location();
        context = c;
```

```
        try {
            IxcRegistry registry = IxcRegistry.getRegistry(context);
            if ( registry == null ) {
                throw new XLetStateChangeException("No registry");
            }
            registry.bind(NAME, location);
        }
        catch(AlreadyBoundException e){
            throw new XletStateChangeException("Something bound");
        }
        catch(StubException e){
            throw new XletStateChangeException("Stub error");
        }
    }

    public void pauseXlet() {
    }

    public void startXlet()
        throws XletStateChangeException {
    }

    public void destroyXlet(boolean b)
        throws XletStateChangeException {
        IxcRegistry registry = IxcRegistry.getRegistry(context);
        if ( registry == null ) {
            throw new XLetStateChangeException("No registry");
        }
        registry.unbindAll();
    }
}
```

The Xlet creates an instance of the `Location` class it wants to remote during initialization, and it uses the `IxcRegistry` for the current `XletContext` to bind it to the name `XletLocationProducer.Location`. This *should* always succeed, but it can fail for one of three reasons:

- The `IxcRegistry` cannot obtain an instance for the `XletContext`. This should never happen, but if it does, the resulting `registry` will be `null`.
- Another Xlet has already bound a remote object to the name `XletLocationProducer.Location`. That might happen if the Xlet is running in more than one context, or if another Xlet uses the same name for something it wants to remote.
- The `IxcRegistry` fails to create a stub for the remoted class.

Of these failures, the most likely is a `StubException`, which can happen if you don't implement your remote class correctly. To implement your remote class correctly, you must ensure that

- The class implement `java.rmi.Remote`
- Each remote object method declare `java.rmi.RemoteException` in its `throws` clause
- The type of each remote object method only accept and return primitive Java types (including, of course, `void`) or those classes that extend `java.io.Serializable`

These restrictions make sense when you think about them, because the IXC mechanism must have a way to know that a remote object is remotable, can handle exceptions encountered during the remote execution, and can pass objects from one Java class loader to another during remote execution.

On the Xlet's exit, the Xlet unregisters the object from the `IxcRegistry` using `unbindAll`. This, too, can fail, but there's little the Xlet can do about these failures at exit time, so the Xlet just logs the failures for debugging purposes. It's not strictly necessary that the Xlet do this—the runtime will take care of it if it doesn't—but it's good practice.

■ **Tip** When your Xlet exits, use `unbindAll` at this point rather than invoke `unbind` for each named object, so that if you add additional objects to the registry, you won't need to add matching `unbind` requests.

Using a Shared Object

At any point, an Xlet can query the `IxcRegistry` for a remoted object by name using the well-known name another Xlet used to register the object. Listing 9-9 shows a hypothetical consumer Xlet that uses the `Location` object shared by the `XletLocationProducer`.

Listing 9-9. *Accessing a Remote Object*

```
import java.rmi.*;
import javax.microedition.xlet.*;
import javax.microedition.xlet.ixc.*;


public class XletLocationConsumer implements Xlet {
    private XletContext context;
    private Location location;
    private static final String NAME =
                            "XletLocationProducer.location";

    public XletLocationConsumer () {
    }

    public void initXlet( XletContext c )
        throws XletStateChangeException {
        context = c;

        try {
            IxcRegistry registry = IxcRegistry.getRegistry(context);
            if ( registry == null ) {
                throw new XLetStateChangeException("No registry");
            }

            location = (Location)registry.lookup(NAME);
        }
        catch(NotBoundException e){
            throw new XletStateChangeException("Nothing bound");
        }
        catch(StubException e){
            throw new XletStateChangeException("Stub error");
        }
    }

    public void pauseXlet() {
    }

    public void startXlet()
        throws XletStateChangeException {
    }
```

```
    public void destroyXlet( boolean b )
        throws XletStateChangeException {
    }
}
```

The process for obtaining a remote object is similar to that for registering a remote object: obtain a reference to the `IxcRegistry` for the `XletContext`, and then invoke `lookup` to obtain the stub for the remoted object. The returned stub will be a proxy to the original class, so you can cast it to the original class definition; you should be sure that both the producing and consuming Xlets include the class definition for the remoted class.

When looking up a remote object, many of the same things can happen as when registering one. In very rare circumstances, there may be no registry with which to obtain a remote object, or the registry may fail to create the local stub for the remote object. It's far more likely, however, that the remote object simply isn't registered, because the producing Xlet isn't running. If that happens, the registry's `lookup` method will throw a `NotBoundException`, and your Xlet will be free to proceed as it sees fit per its business rules without the remote object.

Be prepared for your remote object disappearing while you're using it. When this happens, accessing any of the object's methods throws a `RemoteException`. This can happen for a number of reasons, but the most likely reason is that the Xlet producing the object has exited, and the original remoted object has been freed. For that reason, it's generally best to access a remote object only when you need its services, and release it as soon as you're done using it, rather than relying on the serving Xlet and object to persist throughout the lifetime of the consuming Xlet.

Wrapping Up

The CDC has similar application execution constraints to the CLDC, including limited memory and I/O options. To reflect this, the PBP atop the CDC defines the Xlet, which is an executable interface not dissimilar to the MIDlet defined by the MIDP.

Like the MIDlet, an Xlet has a well-defined life cycle that consists of three states: *paused*, *active*, and *destroyed*. An Xlet begins its life after creation in the paused state, moving to the active state when the application manager calls its `startXlet` method. Similarly, the application manager can pause the Xlet at any time by invoking its `pauseXlet` method, and it can resume execution by forcing the Xlet to re-enter the active state by invoking `startXlet` again. At any time, when the Xlet is in any state, the application manager can terminate the Xlet's execution by invoking the `destroyXlet` method. Unlike a MIDlet, however, the Xlet can refuse a state transition and generate an error by throwing the `XletStateChangeException`.

An Xlet has an accompanying `XletContext` object, which gives the Xlet access to critical information about the application management environment, including the class loader used by the Xlet, the root container into which it should place its GUI elements, and runtime properties. An Xlet can also access any launch arguments provided by the application management system as a property of the `XletContext`.

Multiple Xlets can run at one time; the application manager partitions the screen into separate containers, giving a root container to each Xlet. Unlike the MIDP, which defines a user-interface hierarchy, the PBP requires only the presence of lightweight AWT components that an Xlet and its supporting classes may override to create a user interface. This can pose challenges for cross-platform Xlets, because different Java ME devices supporting the CDC may have different user interfaces, spanning the gamut from custom GUIs based on the lightweight AWT hierarchy to the AGUI implementing much of Java Swing, to other Java packages such as those that provide SVG and 3D graphics.

Xlets can share data through an interface based on the Java RMI; the runtime for PBP-enabled devices includes a system-wide registry for remoted objects that implement the `java.rmi.Remote` interface. Using this interface, you can write classes that implement objects that share data through methods, or you can offload computational tasks to another running Xlet.

CHAPTER 10

Introducing Applets and the Advanced Graphics and User Interface

High-end devices based on the CDC are, in many ways, very close in features to true computers. Many have always-on access to the Internet or to other networks with gateways to the Internet; some may even have built-in web browsers. To reflect these capabilities, the PP defined in JSRs 62 and 216 defines support for applets and provides backward compatibility with the Xlet application model you saw in the last chapter. The applet model defined by the PP is identical to the classic applet model that's been with Java since the beginning; PP devices can execute applets in either an embedded web browser or in a native execution environment that provides the same facilities. Moreover, the PP requires the presence of the Java AWT, making it easy to create complex user interfaces.

In this chapter, I explain the applet execution model supported by devices that provide the PP. I review the fundamentals of writing applets before discussing the user-interface options available to you when writing Xlets and applets for PP-enabled devices. In the process, I discuss both the AWT and Advanced Graphics and User Interface (AGUI) implementations that you may encounter on devices that support the PP.

Writing Applets for Java ME

The original intent of the Java applet execution model was to provide an execution environment for Java code downloaded over the Web within the confines of a web browser. Dating back to the earliest days of Java, the applet adheres to strict security restrictions, including the following:

- An applet cannot define or invoke native methods.
- An applet cannot access files on the host that executes the applet.

- An applet cannot make network connections except to the host that provided the applet.
- An applet cannot invoke any program on the host executing the applet.

These are limitations not placed on Xlets, making the applet execution model more desirable in the context of providing backward compatibility with existing applets or offering a robust web browsing experience than in the context of providing an application execution environment for new applications.

Like Xlets and MIDlets, applets have a life cycle enforced through the interface provided by the base class, `java.applet.Applet`. An applet context accompanies a running applet, permitting applets access to resources such as sounds, images, and other applets running within the same context.

Looking at the Applet Life Cycle

Figure 10-1 shows the states an applet passes through during its execution. Unlike Xlets and MIDlets, an applet cannot pause execution; instead, the applet execution environment can stop or start an applet. Specifically, the runtime can perform the following actions:

- Create the applet, invoking the applet's constructor.
- Give the applet an opportunity to initialize by invoking its `init` method.
- Start the applet by invoking its `start` method.
- Stop the applet when it loses focus, such as when a browser navigates to a different page, by invoking its `stop` method.
- Restart the applet (possibly reloading and reinitializing it) when the applet regains focus by invoking `start` again.
- Stop the applet and give it an opportunity to perform any final cleanup by invoking its `destroy` method before garbage collecting the applet. Of course, an applet can be destroyed at any other time, too.

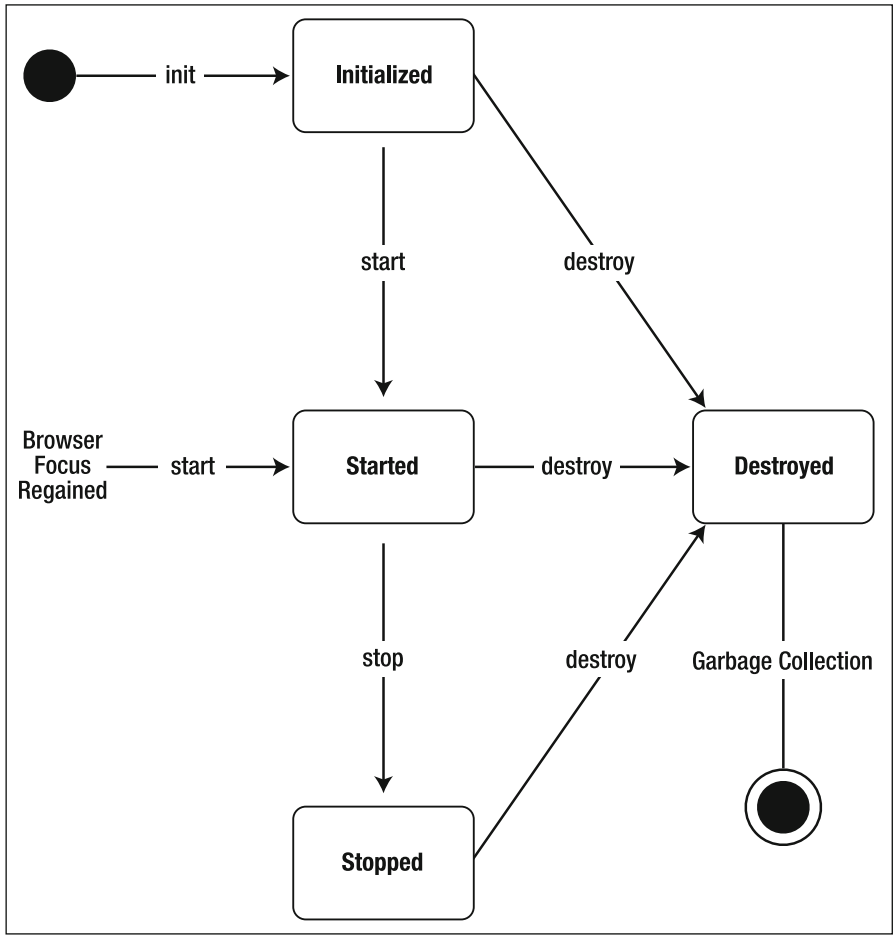


Figure 10-1. *The life cycle of an applet*

Listing 10-1 shows the interface an applet must implement to follow the applet life cycle.

Listing 10-1. *The Applet Interface*

```
import java.applet.*;

public class SimplestApplet extends Applet {
    public void init() {
    }

    public void start() {
    }
```

```
    public void stop() {
    }

    public void destroy() {
    }
}
```

As you can see from Listing 10-1, the applet model is deficient in comparison with the Xlet model in two key ways: it cannot refuse a state transition, and it has no notion of pausing or resuming execution. Note that at an applet does *not* need to override any of the methods in Listing 10-1 if it has nothing to do during a state transition; the class provides default implementations of each.

Presenting the Applet's User Interface

Under the hood, applets are actually containers; specifically, they're subclasses of `java.awt.Container`, meaning that applets can contain other user-interface objects or perform their own painting by overriding the `paint` method. The Hello World applet shown in Listing 10-2 demonstrates overriding the `paint` method, which the window toolkit invokes when the applet is first drawn and whenever it is revealed.

Listing 10-2. *Overriding the paint Method*

```
package com.apress.rischpater.HelloApplet;

import java.applet.*;
import java.awt.Graphics;

public class HelloApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 0, 0);
    }
}
```

Listing 10-3 shows an approach that uses Java AWT components and containers.

Listing 10-3. *Using Java AWT Components and Containers*

```
package com.apress.rischpater.HelloApplet;

import java.awt.*;
import java.applet.*;

public class HelloApplet extends Applet {
    public void init() {
        Label label;
        label = new Label("Hello world!");
        add(label);
    }
}
```

The difference between these two examples lies in how drawing is performed. The code in Listing 10-2 uses the window toolkit's `Graphics` object directly, while the code in Listing 10-3 delegates drawing to components provided by the window toolkit. Because applets are only supported by the PP, you're assured that at least the Java AWT is available on platforms that support applets.

Accessing an Applet's Context

An applet has access to resources on the server that provided the applet, including images and audio clips. These can be fetched directly either by the applet or the applet's context. You can obtain the applet's context (an instance of the `AppletContext` class) by invoking the applet's `getAppletContext` method. The `AppletContext` represents the environment in which one or more applets are executing, such as a web page or Java-based application manager. With an instance of an `AppletContext`, you can perform the following operations:

- Use `getApplet` to obtain a reference to a named applet running within the `AppletContext`.
- Use `getApplets` to enumerate across the list of applets managed by a specific `AppletContext`.
- Use `getAudioClip` to obtain an audio clip from the applet's server, or use `getImage` to obtain an image from the applet's server.
- Use `showDocument` to dispatch a request to view a web page.
- Use `showStatus` to update the context's displayed status.

Because the methods of the `AppletContext` class are clearly browser-centric, and because of the relationship that traditional Java applets have with a web browser, the methods are not as powerful as the context provided to MIDlets and Xlets. One interesting deficiency of the `AppletContext` class when compared with the MIDlet and Xlet contexts is the inability of an applet to request its own termination via the context; instead, an applet that wants to terminate its own execution must perform a `showDocument` request to a URL different than that of the document containing the applet.

Communicating Between Applets

As you might surmise from the `AppletContext`'s `getApplets` method, more than one applet can be running at once, and more than one applet can share the same applet context. This lets multiple applets sharing the same applet perform interapplet communication, as each of the applets sharing a single context can learn of each other's existence and invoke each other's methods. For multiple applets that obtain information from the applet server, you might perform interapplet communication to pipeline requests so that only one applet needs to perform the network activity, for example. In order for applets to communicate with each other, the applets must originate from the same server (and, depending on the implementation, the same *directory* on the same server) and run in the same context—typically, in the same window of the browser or on the same screen.

For one applet to find another, the searching applet must know the sought applet's name. How you specify this depends on how the target device loads applets; if the target hardware uses a web browser as its container for applets, you'll specify the name of an applet using either the `<APPLET>` tag or a `<PARAM>` tag when you specify the source of an applet. For example, you might write the code shown in Listing 10-4.

Listing 10-4. *Specifying the Name of the Applet*

```
<APPLET CODEBASE="applets/" CODE="HelloWorld.class"
    WIDTH=200
    HEIGHT=200
    NAME="hello"/>
```

Listing 10-4 uses the `NAME` attribute of the `APPLET` tag, but you can also use the more extensible `PARAM` attribute of the `APPLET` tag, as you see in Listing 10-5.

Listing 10-5. *Another Way to Specify an Applet Name*

```
<APPLET CODEBASE="applets/" CODE="HelloWorld.class"
    WIDTH=200
    HEIGHT=200>
    <PARAM NAME="name" VALUE="hello"/>
</APPLET>
```

The APPLET HTML tag specifies that the Java applet defined in the Java class file HelloWorld.class should be loaded from the web server's applets directory and given the name hello within the current applet's context.

Returning to the topic of interapplet communication, consider two applets: Producer and Consumer. Producer provides an interface used by Consumer within the context of a single execution environment such as a web page. Producer only needs to declare a method to be used by Consumer, as shown in Listing 10-6.

Listing 10-6. *Declaring a Method Used by Consumer in Producer*

```
public class Producer extends Applet {
    … methods here …
    public void processRequest(String anArgument) {
        … do something with anArgument …
    }
}
```

The HTML that serves the Producer applet must identify Producer so that Consumer can find it by name using the AppletContext, as shown in Listing 10-7.

Listing 10-7. *Identifying the Producer Applet*

```
<APPLET CODEBASE="applets/" CODE="Producer.class"
    WIDTH=200
    HEIGHT=200
    NAME="producer"/>
```

The Consumer applet, which invokes Producer's processRequest method, only needs to locate the Producer in the applet context. It can then treat the resulting object like any other Java object, invoking its processRequest method, as shown in Listing 10-8.

Listing 10-8. *Invoking the Producer's processRequest Method*

```
public class Consumer extends Applet
                      implements ActionListener {
    … methods here …
    public void actionPerformed(ActionEvent e) {
        Applet producer = null;
        producer = getAppletContext().getApplet("producer");
        if (producer != null &&
            producer instanceof Producer) {
            ((Producer)producer).processRequest("Hi!");
```

```
        } else {
            // Handle the error somehow
        }
    }
}
```

In other words, the applet context provides a named registry of applets running in the same context; once you retrieve an applet from the registry, you can simply invoke one of its methods. Of course, good programming practice dictates that you perform some sort of error handling when you do this, either by testing the target explicitly to see if it implements the desired interface for your interapplet communication, or by catching the `NoSuchMethod` exception that may occur if the class you're dispatching against isn't the class you expect. One common reason why this might occur is if the applet manager has loaded one applet but not the other.

Developing User Interfaces with the AWT

Historically, the PP, which is a superset of the PBP, was derived from PersonalJava, Sun's first foray into set-top boxes and other consumer electronics devices. As such, it includes support for most of Java's AWT, including heavyweight components. Expensive in terms of memory and its real-time operating system (RTOS) footprint, including the AWT is required because the PP supports applets that are backward compatible with web applets, as well as permits device manufacturers to host and provide applets for new devices.

■ **Caution** In the name of backward compatibility, the PP supports the deprecated Java 1.0.2 event model, which is based on the `java.awt.Event` class. But that doesn't mean you should use the Java 1.02 event model, because it's deprecated and may disappear in a future version of the PP.

The PP includes most of the AWT classes in J2SE 1.3. The following lists some of the key omissions:

- Two-dimensional graphics classes such as `java.awt.Paint` and `java.awt.Stroke`
- Support for printing through `java.awt.PrintJob` and `java.awt.PrintGraphics`
- Support for accessible interfaces through the `java.awt.Component.AccessibleAWTComponent` hierarchy

It's important to remember that the AWT is only available on CDC platforms implementing the PP; however, the PP includes both the Xlet and applet execution models. In other words, you can leverage the AWT as defined in the PP for both applets and Xlets, provided that your applets and Xlets run on hardware supporting the PP.

Figure 10-2 shows the AWT class hierarchy for user-interface components calling out the major classes in the hierarchy. You can divide the components into the following two key groups: AWT *components* and AWT *containers*. A component, of which `Component` is the superclass, represents any user-interface object. This includes buttons, check boxes, radio buttons, labels, lists, and text-entry items. A container, of which `Container` is the superclass, is a component that can contain other components. Containers have layout managers, which are responsible for where components appear within a container.

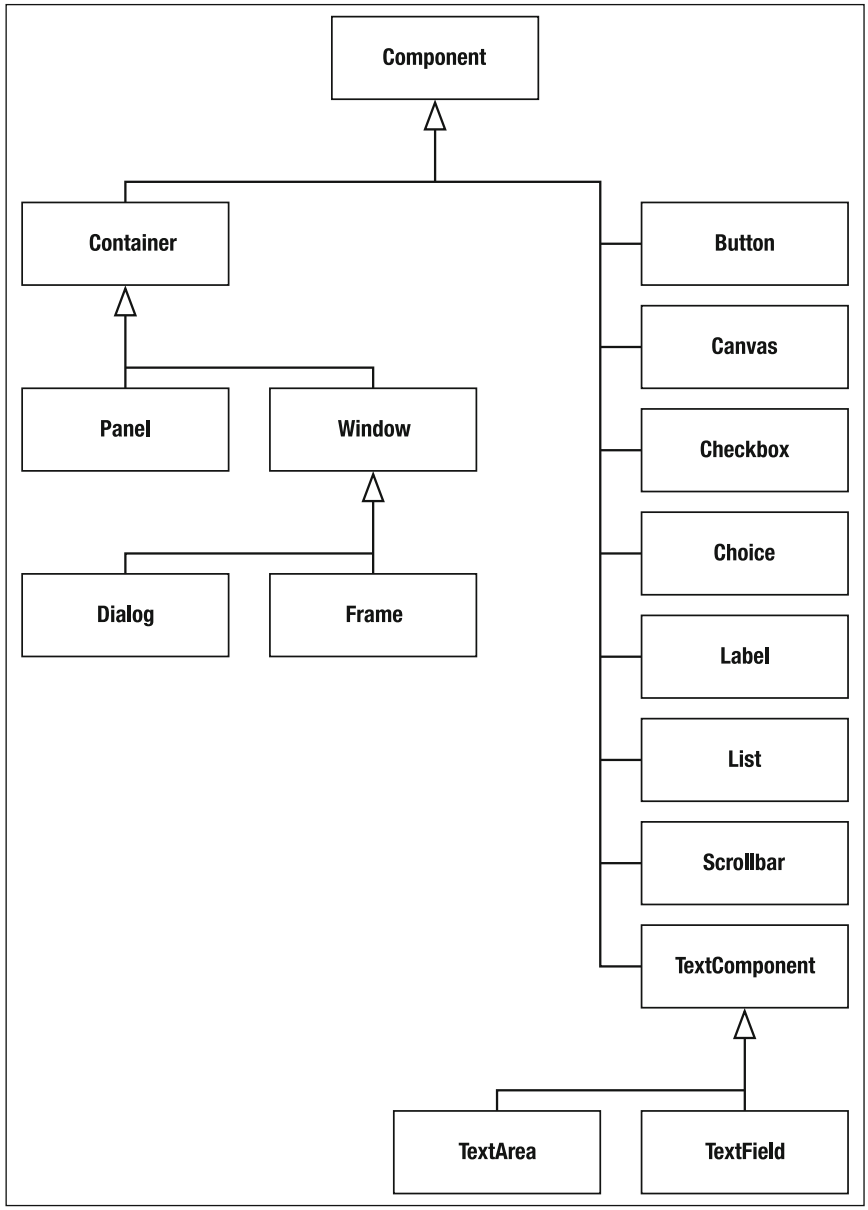


Figure 10-2. *The AWT class hierarchy*

■ **Note** If you need to learn the basics of programming with Java AWT, pick up a copy of *Beginning Java SE 6 Platform: From Novice to Professional* by Jeff Friesen (Apress, 2007) or visit Sun's documentation on the topic at `http://java.sun.com/javase/6/docs/technotes/guides/awt/`.

Using AWT Containers

In AWT, a container is simply a component that can contain other components. The lightweight `Panel` container lets you group related components, while the `Window` and `Frame` containers have corresponding heavyweight implementations in the native platform's window environment. A `Window` provides a top-level window in the native platform's window environment, while a `Frame` provides a top-level title and a border.

A container maintains a list of components and lets you add additional components to that list. The various `add` methods of `Container` let you add components, specifying not just the component to add, but where in the list of components the new component should be, as well as any constraints on how the component should be placed within the container.

The list represents only the components within a container and the Z-order for those components; the container's *layout manager* determines how components are laid out visually within the bounds of the container. The layout manager uses the minimum and preferred sizes specified by each component and the container bounds and an algorithm to determine how the components should appear within the container. The AWT defines five layout managers:

- `BorderLayout`: Arranges its components to fit in five regions: north, south, east, west, and center. When adding components to a container using the `BorderLayout`, you indicate which region the component should occupy; the `BorderLayout` considers the preferred size of each component when positioning each component.

- `CardLayout`: Arranges its components so that each component occupies a virtual card; only one card (and thus one component) is visible at a time. The `CardLayout` defines a set of methods that allow an application to traverse the cards sequentially or show a specific card.

- `FlowLayout`: Arranges components in a left-to-right, top-to-bottom flow.

- `GridBagLayout`: Aligns components vertically and horizontally without requiring the components to be the same size. Each component has a set of corresponding constraints that indicates how the component should be placed in the container.

- `GridLayout`: Aligns components vertically and horizontally in a rectangular grid, requiring each component to be the same size.

When you create a container, its default layout manager is the `FlowLayout`.

You can query a container to obtain one of its components using `getComponent` or `getComponentAt`; the former method accepts an index into the list of components, while the latter takes either a `Point` or *x* and *y* coordinates and returns the component at the specified location. You can also test if a container contains a specific component by invoking the container's `isAncestorOf` method.

Typically, you use containers to group components during the creation of your user interface, which itself may inherit from the `Panel` or `Window` classes. The typical pattern is to create a container's components in its constructor or in a method that its constructor invokes, such as an `initComponents` method. Because creating user interfaces by hand can be tedious, consider using something like NetBeans, which provides both a source and design view of `Container` classes.

■ **Tip** In NetBeans, select New ➤ JPanel Form… or New ➤ JFrame Form when right-clicking a package in the Projects pane, then choose the Design view for the new document. You can change the base class of your container in the Source view to be a `java.awt.Panel` or `java.awt.Frame`, and the palette NetBeans provides includes the AWT components just below the Swing components.

Because of the limitations that many consumer electronics devices have, the relationship between the `Window` class and the underlying window may be somewhat tenuous. This is because the location, size, and label of top-level windows are under the control of the native window manager. The methods that let you adjust a container's title, bounds, and position provide guidelines to the native window manager, rather than absolutes that the manager must obey. As a result, the window manager may ignore these requests or modify them in order to present the `Window` appropriately. On some platforms, these may be asynchronous operations; interfaces such as `getLocation`, `getSize`, and `getLocationOnScreen` may not reflect the actual geometry of the `Window`. Moreover, the window manager may ignore the decoration, title, and resizability of a `Frame`.

Using AWT Components

The various subclasses of `java.awt.Component` are where your user interface's rubber meets the road. As you saw in Figure 10-2, the hierarchy of components includes the basic widgets you need to create most user interfaces, and you can always create your own either by composing several components in a container or by drawing a component's contents and handling events directly.

The `java.awt.Component` interface manages several different things:

- The attributes of a component, such as a component's position, font, visibility, opacity, and so forth
- Event handling for the component; these methods comprise a deprecated set of methods now replaced with the listener model I discuss in the "Handling AWT Events" section later in this chapter
- Methods to support drawing the component
- Methods that manage focus and indicate which component has focus

As you have already seen, AWT components can come in two flavors: heavyweight and lightweight. Heavyweight components are *proxy* interfaces to a *peer* object in the native window toolkit; for example, if the `java.awt.Button` class has a native component corresponding to the button, there will be a peer class somewhere that encapsulates the interface to the native button. Lightweight components, on the other hand, have no corresponding tangible entity in the native window toolkit; the Java environment renders them on the fly. Although not a requirement, it's likely that if you encounter the AWT on a PP-compliant device, the AWT will implement an interface to heavyweight components. Heavyweight components have an advantage in that an application developed with heavyweight components can generally match the look and feel of the native platform, because the controls used by the heavyweight component toolkit are the same controls as those provided by the underlying platform. However, the implementation of the classes that bridge the AWT with the underlying platform can consume valuable heap and ROM space, which is why not all profiles for the CDC include heavyweight components.

When working with components, it's a good idea to avoid mixing lightweight and heavyweight components, due to limitations both in the AWT and the underlying platform. Lightweight components—including those in the AGUI, which I discuss later in this chapter's "Developing User Interfaces with the AGUI" section—have support for transparent regions, unlike a heavyweight component. Among other things, this means that heavyweight components occlude what's behind them and can only appear rectangular. Moreover, when heavyweight and lightweight components intersect, the heavyweight component is always on top, regardless of Z-order. Finally, a heavyweight object consumes any mouse events that fall on it; a lightweight component passes events on to its parent container.

Handling AWT Events

With the AWT, you handle events by overriding specific methods that receive events from the window toolkit. You can do this in one of two ways: by intercepting the low-level events that are passed to a component, or by registering listeners for high-level events.

Handling low-level events is best done by specific components, such as when writing new lightweight components (a topic I discuss in Chapter 9). If you need to process low-level events, you can do so by following these steps:

1. Enable receipt of the appropriate events by invoking `enableEvents` and passing a constant for the kind of events the applet wants to receive.

2. Override either `processEvent` or one of its delegates (`processComponentEvent`, `processFocusEvent`, `processKeyEvent`, `processMouseEvent`, `processMouseMotionEvent`, `processInputMethodEvent`, or `processMouseWheelEvent`) to handle any incoming events.

3. Disable event receipt by invoking `disableEvents` when you don't want to receive any events.

By far the simplest way to handle events, however, is to extend the component to include a specific listener for a high-level event, and then implement the corresponding listener. The Java event model—used by both the AWT and Swing, which is the basis for the AGUI—includes listener interfaces for a variety of events, including the following:

- `ActionListener`: Handles high-level actions generated by components
- `FocusListener`: Handles focus changes in the `java.awt.Container` and `Container` subclasses
- `KeyListener`: Handles individual keyboard events
- `MouseListener`, `MouseMotionListener`, and `MouseWheelListener`: Handle mouse- and pointer-generated events
- `TextListener`: Handles events that indicate when a text field's contents have changed

Listing 10-9 shows a simple applet that responds to button clicks on its button.

Listing 10-9. *Responding to Button Clicks*

```
package com.apress.rischpater.buttonsample;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
public class ButtonApplet extends Applet
                          implements ActionListener {
    Label label;
    public ButtonApplet() {
    }

    public void init() {
        label = new Label("Hello World");
        Button button = new Button("Click Here");
        add(label);
        add(button);
        button.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        label.setText("Click!");
    }
}
```

While simplistic, this code shows the basic idea: objects interested in receiving events must declare themselves as implementing the appropriate listening interface, and then signal to the generator of the event that they're listening for events by invoking the appropriate method. This example does this by implementing the `ActionListener` contract that provides the `actionPerformed` method. Additionally, the applet notifies the button, which generates `ActionEvents`, that it wants to receive any `ActionEvents` that the button generates.

Developing User Interfaces with the AGUI

The AGUI package, defined in JSR 209, provides a subset of the Java Swing classes for Java ME devices. Using the AGUI, you can develop essentially a single application that runs atop both a AGUI-enabled Java ME device and a conventional computer. The AGUI package depends on CDC 1.1, FP 1.1, and PBP 1.1, and it provides the following packages:

- A subset of the `java.awt` package to provide the class hierarchy for implementing the AGUI's lightweight components
- The `java.awt.font` package to support rendering PostScript Type 1, Type 1 multiple master, OpenType, and TrueType fonts
- The `java.awt.geom` package for operations on objects related to two-dimensional geometry

- The `java.awt.image` package to support streamed creation and modification of images; the AGUI requires support for GIF89a, JPEG, JFIF, and PNG version 1.0 images
- The `java.nio` package for specifying byte order
- The `javax.imageio` package, along with `javax.imageio.event`, `javax.imageio.metadata`, and `javax.imageio.stream` to support image I/O
- The `javax.microedition.plaf` package, which differs from the Java SE implementation in that the implementation APIs are not classes but rather interfaces to permit OEMs and service providers to deeply customize the look and feel of the AGUI
- The `javax.microedition.agui.event` class to support the dedicated hardware keys typically found on consumer electronics devices
- The `javax.swing` package and its subpackages `javax.swing.border`, `javax.swing.event`, `javax.swing.plaf`, `javax.swing.table`, `javax.swing.text`, `javax.swing.tree`, and `javax.swing.undo` to provide lightweight components that work the same across multiple platforms

Using the AGUI is similar to using the AWT, in that you organize your user interface around containers that contain components, and you register event listeners to handle user-interface events. Two key differences arise, however, especially when using some of the more sophisticated components provided by Swing. First, Swing's architects designed Swing's components around the MVC design pattern. Understanding MVC can make interacting with the more complex Swing components easier. Second, Swing supports a pluggable look and feel (although the AGUI support does *not* provide the same functionality).

■ **Note** If you need to learn the basics of programming with Java Swing, as with AWT, I suggest you pick up a copy of *Beginning Java SE 6 Platform: From Novice to Professional* (Apress, 2007) by Jeff Friesen or visit Sun's documentation on the topic available at `http://java.sun.com/javase/6/docs/technotes/guides/swing/`.

Using the AGUI is similar enough to Swing that with your existing Swing experience, you should feel right at home. However, there are some differences, including the following:

- Possible limits on the behavior of containers requiring corresponding top-level windows
- Additional support for input constraints and device-specific keys
- Restrictions on subclassing drawing behavior for components

In addition, several class methods have some optional restrictions depending on the AGUI implementation; your application can query the system by means of a system property to determine the actual behavior. Table 10-1 shows a list of the system properties you can query and their meaning; the subsequent sections describe the AGUI limitations in more detail.

Table 10-1. *The AGUI System Properties and Their Values*

Property	Value
`javax.swing.JComponent.setBackground.isRestricted`	`true` if and only if `JComponent.setBackground` is restricted by the native window toolkit
`javax.swing.JComponent.setBorder.isRestricted`	`true` if and only if `JComponent.setBorder` is restricted by the native window toolkit
`javax.swing.JComponent.setForeground.isRestricted`	`true` if and only if `JComponent.setForeground` is restricted
`javax.swing.JComponent.setToolTipText.isRestricted`	`true` if and only if calls to `JComponent.setToolTipText` fail silently because tool tips are not supported
`javax.swing.JList.setCellRenderer.isRestricted`	`true` if and only if `JList.setCellRenderer` cannot set custom cell renderers
`javax.swing.setMnemonic.isRestricted`	`true` if and only if calls to `setMnemonic` fail silently
`javax.swing.text.JTextComponent.setFocusAccelerator.isRestricted`	`true` if and only if calls to `JTextComponent.setFocusAccelerator` fail silently
`javax.swing.JMenuBar.clientPropertiesSupported`	`true` if and only if `JMenuBar` supports client properties
`javax.swing.JMenuBar.NUM_SOFT_KEYS`	The number of soft keys available on the device, if `JMenuBar` client properties are supported; undefined otherwise
`javax.swing.JMenuBar.NUM_SOFT_MENUS`	The number of soft menus available on the device, if `JMenuBar` client properties are supported; undefined otherwise
`javax.swing.JTabbedPane.setToolTipTextAt.isRestricted`	`true` if and only if calls to `JTabbedPane.setToolTipTextAt` fail silently

Finally, a gentle reminder: the AGUI, like Swing, is *not* thread-safe. Swing components should be accessed by only one thread at a time; usually, this is the event-dispatching thread. If you're writing a multithreaded application that needs to work with the AGUI interface, you should ensure that all AGUI updates occur on the event-dispatching thread. You can do this using the `javax.swing.SwingUtilities` class; it provides the static methods `invokeLater` and `invokeAndWait`, which take a `Runnable` to perform on the event-dispatching thread after the window system handles all pending events. Use the `invokeLater` method to queue an action to perform on the event-handling thread; this is preferable to `invokeAndWait`, which blocks until the window system has completed your task.

Understanding Restrictions on Top-Level Windows

The PBP permits devices to disallow multiple top-level windows. When the AGUI is combined with a device that does not permit multiple top-level windows, the AGUI suffers from the same limitation: only a single `JFrame` is permitted on a single graphics device. Attempts to construct a second `JFrame` fail, and the runtime throws the `java.lang.UnsupportedOperationException`. This means that, for maximum portability between AGUI-enabled devices, your user interface should reside entirely in one window.

This has ramifications for other containers as well. For example, the `JOptionPane` may only use lightweight containers to display dialogs, and it may not be possible to create a top-level dialog by creating a `JOptionPane` with a `null` value for its parent component. Similarly, pop-up menus should be rooted in a specific container; you can't create a `JPopupMenu` with a `null` invoker. Moreover, like the `JOptionPane`, the `JPopupMenu` may only use lightweight components in its composition.

Using the AGUI's Added Input Support

Many consumer electronics devices do not have a traditional keyboard; in fact, many consumer electronics devices have *different* input devices from each other altogether. This poses a challenge for any generic window toolkit, which must somehow abstract the differences between disparate devices.

The AGUI borrows the notion of input constraints that limit what a user can enter from MIDP 2.0 by passing the `INPUT_CONSTRAINT` client property to a `JTextComponent` via the `putClientProperty` method. Table 10-2 shows the input constraints that an implementation of the AGUI may support.

Table 10-2. *AGUI-Supported Input Constraints and Values*

Constraint	Numeric Value Passed to putClientProperty	Description
ANY	0	The user is allowed to enter any text.
EMAILADDR	1	The user is allowed to enter an e-mail address.
NUMERIC	2	The user is allowed to enter an integer value.
PHONENUMBER	3	The user is allowed to enter a phone number.
URL	4	The user is allowed to enter a URL.
DECIMAL	5	The user is allowed to enter a signed decimal value.

Because the AGUI may be implemented on devices with specific hardware keys—such as a remote control or physical buttons on a PDA or smartphone—the AGUI defines the `javax.microedition.agui.event.DeviceKeyEvent` that specifies a hardware key event such as a keypad volume control, dedicated application launch facility, or media control such as rewind or fast forward. Key event listeners may receive key events (instances of subclasses of the `java.awt.event.KeyEvent` class) with the value `VK_UNDEFINED`, and then test the event using `instanceof` to determine if the `KeyEvent` is really a `DeviceKeyEvent`. If it is, you can then determine which hardware key the user pressed by invoking the `DeviceKeyEvent`'s `getDeviceKeyCode` method.

Finally, many consumer electronics devices support *soft keys* that an application can assign a custom label and action. The AGUI lets you define these using the interface to `JMenuBar`, specifying a *menu type* and a *menu priority*. The menu type indicates the intent of the action your application offers through the soft key, and the menu priority indicates the importance of this command label relative to other command labels on the same screen.

Understanding Changes to the Drawing Algorithm

A key compromise of the AGUI is to support the Swing APIs regardless of whether they're implemented as lightweight components or heavyweight components. To meet this goal, platform vendors may make a number of changes to better support the native rendering pipelines found on many of the platforms that implement the AGUI.

Perhaps most noticeable is that the rendering of `JComponent` subclasses is double-buffered by default. You can determine whether double buffering is active by calling the `isDoubleBuffered` method of any `JComponent` implementor. You can also request a component to perform single buffer drawing by calling `setDoubleBuffered` and passing `false`, but the AGUI implementation can choose to ignore your request.

The underlying UI toolkits on many consumer devices are considerably less flexible than that found supporting today's Java SE implementations. To meet these restrictions,

the AGUI limits the ability of applications to render on top of components. Because components may be heavyweight, the `JComponent` paint methods (`paint`, `paintComponent`, `paintBorder`, and `paintComponents`) have been made final on the first concrete subclass of `JComponent`. This is because the AGUI provides optimized, tight integration with the underlying platform, and may use the platform's window toolkit to implement the AGUI.

Another way to look at this restriction is in light of the pluggable look and feel (PLAF) provided by Swing and the AGUI; in Swing, it's easy for developers to override the presentation of Swing components to provide their own look and feel. In the AGUI, this functionality is primarily for platform vendors implementing the AGUI to be able to provide an optimal look and feel for the device, not for developers like you and me to create a specific look and feel.

Wrapping Up

To remain compatible with the large number of applets on the Web and to provide a familiar programming model, the PP provides support for the applet execution model. Identical to the applet model in use in today's web browsers, the applet model that CDC devices provide run applets in the context of an embedded web browser or another execution environment enforcing the same security model.

Applets have a life cycle similar to, but not the same as, MIDlets and Xlets. Notably, there's no defined way to pause and restart an applet, and unlike both MIDlets and Xlets, an applet cannot explicitly ask its containing application to terminate the applet's execution. Because an applet is a subclass of `java.awt.Container`, it can contain other user-interface components or simply override its `paint` method and draw its user interface directly.

To support applets, the PP also supports most of the AWT, which was Java's first user interface hierarchy. While the AWT implementation is not complete—it lacks support for printing, accessible interfaces, and some two-dimensional graphics—it has relatively few limitations. Like the Java SE AWT, the PP-provided AWT is usually implemented using heavyweight components, giving you the ability to create complex user interfaces that reflect the look and feel of the native platform hosting the Java environment.

The AGUI, which is an optional package requiring the CDC, FP, and PBP, provides yet another way for you to develop rich user interfaces on some devices. You can use the AGUI to write not only applets, but Xlets as well. Based on the ubiquitous Swing hierarchy of user-interface classes, the AGUI has a handful of limitations, such as limits on top-level containers and how components can customize drawing behavior. The AGUI also offers a more robust event framework, supporting both soft keys and hardware buttons commonly found on today's consumer electronics devices.

CHAPTER 11

Using Remote Method Invocation

The CDC is expressly for devices with always-on or nearly always-on connections to the Internet. CDC devices offer great potential for network programming, because the demands of limited resources are balanced by a reliable network connection. Java's support for Remote Method Invocation (RMI) provides a programming model that is a welcome alternative to the lightweight model provided by web services. (Ironically, the RMI model in large part predates the move to web services, as it's based on the remote procedure call semantics developed at Sun and popularized in a variety of remote computing initiatives.)

In this chapter, I first give you a whirlwind tour of the Java RMI facility as it is implemented in the Java SE. After presenting you with the high-level architecture and approach to Java RMI, I introduce the key Java RMI interfaces and show how those interfaces fit together to meet two key goals of many networked applications: remoting Java computation, and remoting access to Java objects. Next I turn attention to JSR 66, the Java RMI Optional Package (RMI OP) that defines the implementation of RMI available on some CDC-enabled devices. After I show you what is available through the Java RMI OP, I present an example to help cement the concepts you encounter in this chapter.

Understanding Java RMI

A key feature of the Java programming language is its ability to support distributed programming—applications that run on separate hosts, passing objects back and forth—through the Java RMI paradigm.

All distributed object-oriented applications need to perform three kinds of tasks:

- *Remote object discovery*: Applications need mechanisms to discover references to remote objects.
- *Remote communication*: Applications running in different contexts on different machines need to be able to communicate about the changing state of objects.
- *Remote behavior definition*: Applications must also be able to communicate about the definitions from which those objects are derived.

Let's see how the architecture of Java RMI accommodates these requirements.

Understanding the Architecture of Java RMI

Java RMI meets the needs of distributed programs in two key ways: by providing a remote discovery registry, and by providing a robust communications mechanism for exchanging information about Java classes and objects. Figure 11-1 shows the relationship between the client and the server in an RMI-based application.

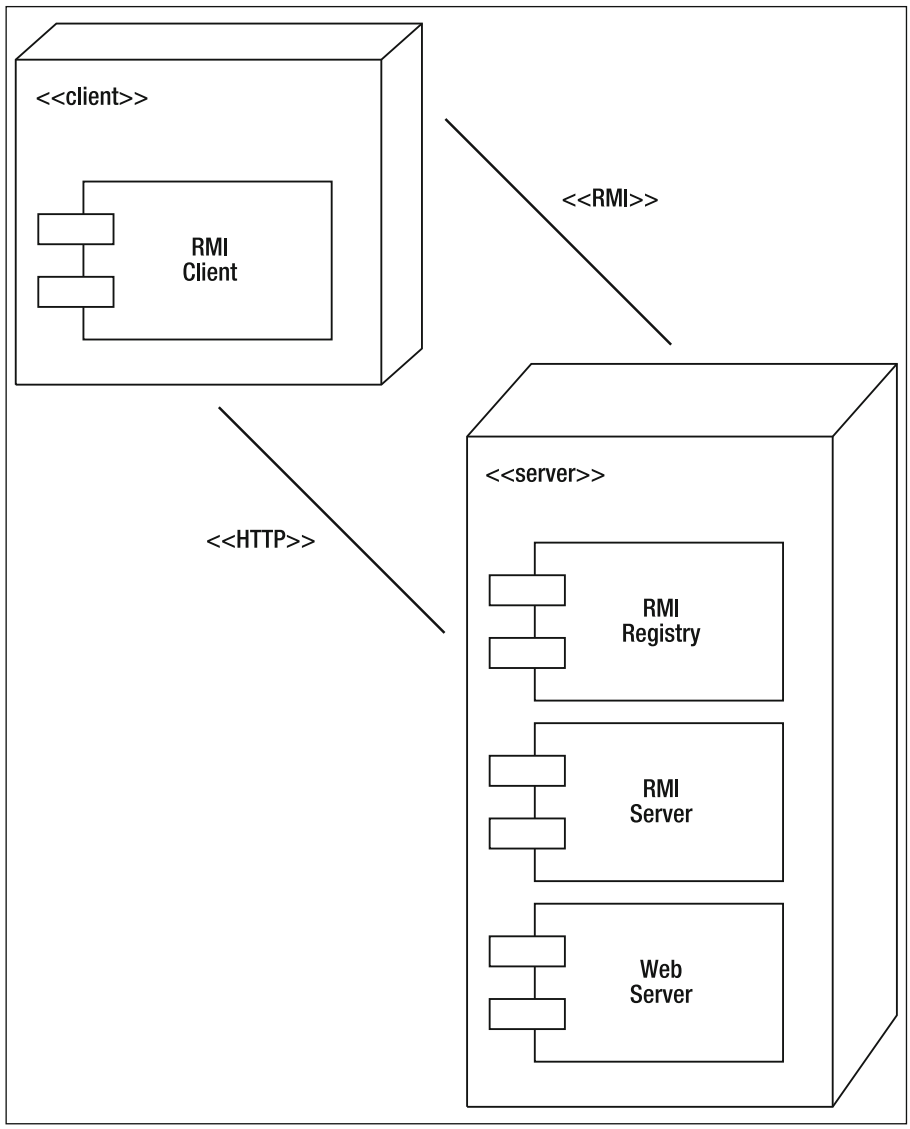


Figure 11-1. *RMI application architecture*

RMI-based solutions have several key components:

- *RMI client application*: This runs within the client's JVM. Supporting the RMI client application is the Java Remote Method Protocol (JRMP).[1]
- *RMI registry*: Maintained by the application server, this provides object discovery to RMI clients.
- *RMI server*: Hosted by the application server's JVM, this is responsible for executing remote objects.
- *Web server*: This is responsible for serving new code to the RMI client application.

You build a distributed application using RMI as you would any other Java application: through interfaces and classes. In your distributed application, some objects are *remotable*—that is, you can invoke a remotable object's methods across JVMs through the protocol provided by the RMI. You gain references to these objects using the RMI registry running on a well-known server; the registry returns a *stub* that implements the methods provided by the remote object. Under the hood, a remote object isn't the same as a local object; instead, it's a proxy, implementing network interfaces that communicate with the remote JVM's instance of the object. Stub objects implement the same interface that the remote object implements, meaning that it can be cast to any of the objects implementable by the remote object. Figure 11-2 shows this relationship; the remote object is the solid circle in the RMI server, and the stub is the dotted circle in the RMI client.

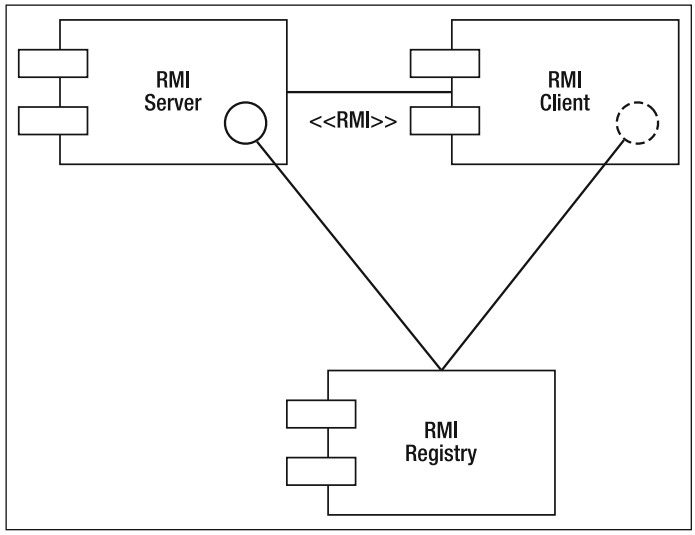


Figure 11-2. *The relationship between remote and local objects in the RMI implementation*

1. Other protocols, such as the Java RMI over Internet Inter-Orb Protocol (RMI-IIOP), as well as proprietary implementations can support RMI; these are not supported by the CDC and are not discussed here.

In truth, there's more to RMI than your interface, implementation, and a stub. Early versions of RMI required both stubs and *skeletons*, which are server-side classes that bear the responsibility of communicating between the stub and the served object. You built these skeletons, like the stubs, using the RMI compiler `rmic`. Today, with the rise of the Java reflection interfaces, RMI applications need only stubs; in fact, as you'll see later in this chapter, not even stub classes are required when using the RMI package tailored for Java ME.

The Java RMI model has its advantages over proprietary and traditional web services approaches, including the following:

- *Object orientation*: The RMI model extends the Java execution model in a seamless way, letting you pass Java objects across the RMI interface. This lets you continue to use object-oriented techniques and design patterns when constructing your application.

- *Security*: The RMI model leverages Java's existing security model.

- *Legacy support*: Servers providing RMI services can connect to legacy services through JNI calls, letting you wrap legacy systems in Java interfaces and integrate them into your networked applications.

These advantages make RMI attractive in many situations, especially with consumer electronics applications in the set-top box market, where security and integration with large-scale legacy computing has been an ongoing effort since the late nineties. At the same time, RMI is somewhat dated, having originated in the mid to late nineties. In comparison with its chief competitor, the XML- and HTTP-based web service model, it can be seen as lacking, especially in the following areas:

- *Interoperability with web services*: The web services model has become the leading client/server architecture pattern in many markets. Connecting an RMI application to an existing web service requires additional work similar to that required by connecting an RMI application to any other legacy application.

- *Weight*: RMI comes with a size penalty on the platforms that support it; RMI requires the CDC and the Foundation Profile for mobile devices. This requires significantly more robust hardware than does the Java ME web services implementation, which can run atop a CLDC platform. In addition, the RMI protocol can be significantly more expensive over wireless links than a carefully designed web service exchange.

- *Network support*: Although the full RMI implementation supports automatic HTTP tunneling through firewalls, as you'll see in the section "Seeing What's Provided by the Java RMI Optional Package" later in this chapter, the Java RMI OP does not. Thus, to support Java RMI on Java ME devices, your network security policy requires additional thought over that required when simply deploying a web service.

Despite these limitations, Java RMI may be a good fit when developing your application, especially if your situation meets one of two criteria. First, clearly, if your application infrastructure already uses RMI, you can best integrate with it by continuing to use RMI throughout your development. This is especially true for some segments of the telecommunication consumer electronics marketplace, where middleware solutions to constrained television set-top boxes are common. Second, if you are developing an application with strong remote computation requirements, RMI provides a better layer of abstraction for expressing programs about remoting computation through agents than a traditional web service.

Introducing the Java RMI Interfaces

The key to understanding Java RMI is understanding the concept of interfaces. As you already know, an interface is a contract: it makes a promise that a particular object will carry out particular functions (the methods of the class), but it doesn't say how it will carry out those functions. In other words, *an interface separates the definition of behavior from the implementation of behavior*. This is key, because RMI uses this concept throughout its implementation; your remote objects are defined in terms of their interfaces, and their implementation (that is, their ability to operate as remote objects) occurs behind your program's back.

The Java RMI interfaces are defined in the `java.rmi` package hierarchy. This hierarchy defines one interface, three classes, and a bunch of exceptions:

- *The* `java.rmi.Remote` *interface*: Specifies the intent of an interface destined for remote execution
- *The* `java.rmi.MarshalledObject` *class*: Abstracts a byte stream with the serialized representation of an object
- *The* `java.rmi.Naming` *class*: Provides the interface to storing and obtaining references to remote objects in a remote object registry
- *The* `java.rmi.RMISecurityManager` *class*: Defines a default security manager used when downloading code from a remote server
- *Exceptions such as* `java.rmi.RemoteException`: Occur when dealing with errors that can occur in a distributed environment

■ **Note** A parallel implementation of Java RMI provides RMI over Internet Inter-Orb Protocol (RMI-IIOP). Now part of Java SE and in the `javax.rmi` hierarchy, it has no corresponding support in Java ME.

To implement the server side of a distributed system, some additional code is necessary. The `java.rmi.server` hierarchy provides this code, including the `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable` classes. The difference between the two classes is that remote objects extending `UnicastRemoteObject` require a server to run for the lifetime of the object, while an object extending `Activatable` runs in the context of Java's RMI daemon `rmid`.

Understanding the Java RMI Optional Package

The Java RMI OP, formalized in JSR 66, came about fairly early in the history of mobile Java. Approved in 2000, it arose from efforts by Motorola, Siemens, Sun, and others to bring RMI to mobile platforms running J2ME, the predecessor to Java ME. These partners developed RMI OP to meet the needs of highly connected, distributed applications that leverage Java standards for information passing. Today, Sun provides a reference implementation of Java RMI OP implemented entirely in Java to licensees that can be integrated directly onto hardware that meets the minimum requirements. RMI OP is interoperable with the Java SE RMI implementation, making it ideal for integrating embedded devices with existing RMI-based solutions.

Looking at the Requirements for the Java RMI Optional Package

Although carefully streamlined to meet the needs of embedded devices, RMI OP still has some sizable requirements, including the following:

- 2.5MB of available ROM over the existing requirements for Java ME and other platform concerns
- 1MB or more of available RAM over the existing requirements for Java ME and other platform concerns
- TCP/IP connectivity to the network
- Support for the Java ME Foundation Profile running atop the Java ME Connected Device Configuration

Many device vendors have moved toward providing support for the CDC on mobile and embedded devices, although few still provide the horsepower necessary to run RMI OP. As the cost of memory and processors continues to fall and we see greater convergence between personal computers and other consumer devices, it's likely that this will change, and the RMI OP will move from being available only in specialized environments to being more generally available.

Seeing What's Provided by the Java RMI Optional Package

How does RMI OP differ from RMI as provided by Java SE? It's first useful to determine how RMI OP is the *same* as Java SE, because it's likely that you will design your solution around Java SE RMI and use a Java SE– or Java EE–based server environment to support your distributed application.

By definition, RMI OP must support the following features:

- Full support for the RMI call semantics, including support for the RMI wire protocol
- Support for marshalling—that is, support for representing an object on a remote host
- The ability to export remote objects from RMI OP–enabled devices through the `UnicastRemoteObject` class
- Distributed garbage collection and garbage collector interfaces across both client and server
- Interfaces to support objects that don't persist between invocations and require a full-time server to manage their activation
- Support for the registry interfaces and the ability for RMI OP–enabled devices to export registry objects remotely

RMI OP is surprisingly comprehensive for a standard aimed at low-end devices. By supporting `UnicastRemoteObject`, distributed garbage collection, and the ability to register and export a remote object, RMI OP–enabled hardware can fully participate in distributed computation, acting as both clients and servers in the object-distribution process. This symmetry is a key feature of RMI OP over the traditional web service model of computation, both because it's required for full interoperability support with Java SE RMI and because it lets embedded devices serve objects to remote servers. In fact, on devices providing RMI OP, RMI can be used for interapplication communication on the same device or for truly distributed applications across the network.

So what's missing? A lot, actually, but odds are that you won't miss most of it:

- Support for RMI request proxying over HTTP
- Support for the RMI multiplexing protocol, which enables multiple JVMs to invoke remote methods on each other when only a single communication channel exists
- Implementation of the `Activatable` interface, providing a concrete implementation of objects that can be activated
- Deprecated methods, classes, and interfaces (which you shouldn't be using anyway!)
- Support for the RMI skeleton/stub protocol, along with the need for the stub and skeleton compiler

These deficits have some practical consequences, but not as many as you might think. First, and most limiting, is that RMI OP can only communicate using the original TCP/IP socket-based RPC mechanism: there's no support for tunneling through firewalls. This has a direct impact on the network topology under which you deploy your application, because you must configure the firewalls that protect your servers to permit Java RMI traffic. The loss of a concrete implementation of `Activatable` may be an issue for you if you want the device to host objects that do not persist between activations, but you're free to provide your own implementation of the interface. And not needing to construct stub and skeletons is actually a blessing; you only need to do this work for your Java SE applications.

Applying Java RMI

When writing a distributed application with Java, you must perform the following steps in concert with the `java.rmi` package:

1. Write the Java interfaces for the remote services.
2. Implement the remote services, deriving your implementation from `UnicastRemoteObject` or `Activatable`.
3. Generate stub classes from the implementation if you're deploying part of the solution on Java SE or Java EE.
4. Write a remote service host container that creates and manages your remote services.
5. Write your Java client, invoking the remote objects on the client.
6. Install and deploy the client and server.

While this sounds like a lot of work, it frees you from many of the low-level details of developing a remote application. Steps 1 and 2 are writing Java code—something you already know how to do. The Java SE comes with the `rmic` compiler to perform step 3. Step 4 is simply writing a small Java application, and step 5 you have to do anyway—your client application is what it's all about. Step 6 can be a small task or a big task, depending on the nature of your application and service.

Let's walk through these steps one at a time using the weather example we've visited throughout the book. Our WeatherApplet example for the CDC needs a source for weather data, presumably sourced from a commercial or governmental service. In the sections that follow, I walk you through building a remotable `Location` class suitable for communicating location and weather data using RMI OP.

To implement the remotable `Location` object, you need to define the relationship between the various interfaces and implementations necessary. Figure 11-3 shows the relationship between the `Location` interface, the RMI OP hierarchy, and the implementation of `Location`, `LocationImpl`.

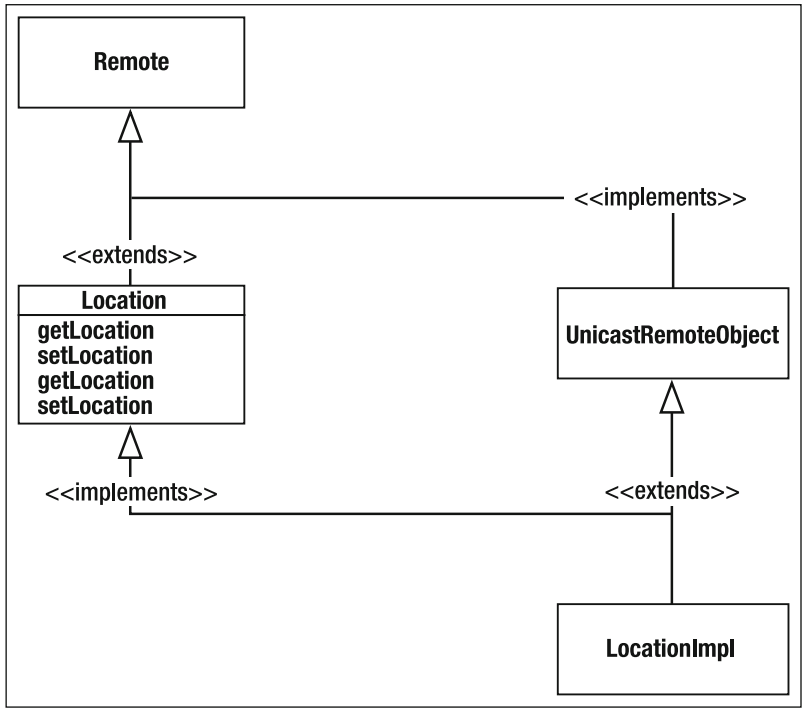


Figure 11-3. *The remotable Location interface and its implementation*

The `Location` interface—which I first introduced as a class for representing spatial locations and weather reports for specific locations in Chapter 6—has two `String` properties exported as properties via the four methods `getLocation`, `setLocation`, `getForecast`,

and `setForecast`. The `Location` interface is remotable, indicated by its extension of the `java.rmi.Remote` interface. Its concrete implementation, `LocationImpl`, is based on the RMI OP implementation of a `java.rmi.UnicastRemoteObject`, which can be shared across a distributed computing environment.

■ **Note** Because NetBeans doesn't directly support the development of RMI applications, the examples that follow require you to have the JDK installed, which should have been installed when you installed NetBeans. You may need to be sure that the `bin` directory of the installed JDK is in your path to access commands such as `javac` and `rmic`.

Writing the Java Interfaces for the Service

As I noted previously, the key to RMI is defining the interfaces to the remote service. Listing 11-1 shows the `Location` interface.

Listing 11-1. *The Location Interface*

```
public interface Location
    extends java.rmi.Remote {
    public String getLocation()
        throws java.rmi.RemoteException;

    public void setLocation(String l)
        throws java.rmi.RemoteException;

    public String getForecast()
        throws java.rmi.RemoteException;

    public void setForecast(String f)
        throws java.rmi.RemoteException;
}
```

This interface is straightforward, although you can see that each method can throw `java.rmi.RemoteException`. This is because implementations of the interface can be proxies, remote from the actual implementation. In that case, the implementation of the interface must have a way to signal an exceptional situation, such as a loss of network connectivity.

Implementing the Service Using Java SE

The service itself is the `LocationImpl` class, which provides a concrete implementation of the `Location` interface with help from the `UnicastRemoteObject`, as shown in Listing 11-2.

Listing 11-2. *The LocationImpl Class*

```
public class LocationImpl
    extends
        java.rmi.server.UnicastRemoteObject
    implements
        Location {
    private String location;
    private String forecast;

    public LocationImpl()
        throws java.rmi.RemoteException {
        super();
    }

    public String getLocation()
        throws java.rmi.RemoteException {
        if (location != null) {
            return location;
        } else {
            return "";
        }
    }

    public void setLocation(String l)
        throws java.rmi.RemoteException {
        location = l;
    }

    public String getForecast()
        throws java.rmi.RemoteException {
        if (forecast != null) {
            return forecast;
        } else {
            return "";
        }
    }
```

```
    public void setForecast(String f)
        throws java.rmi.RemoteException {
        forecast = f;
    }
}
```

Note this implementation's default constructor, which must invoke its superclass's constructor immediately. This is because the class must invoke the `UnicastRemoteObject`'s constructor to perform the linking to the RMI subsystem and remote object initialization.

`LocationImpl` extends the `UnicastRemoteObject`, which provides the necessary links to the RMI subsystem. This is one way to link an object to the RMI subsystem and prepare it for exporting to remote systems; another way is to invoke the `UnicastRemoteObject.exportObject` method, passing an object that implements `Remote`. In either case, the `UnicastRemoteObject` implementation exports the object on a port to make it available to receive incoming method invocations.

■ **Note** A complete implementation of `Location` would use another remote computing interface—say a web service—to obtain a remote location's weather forecast. For brevity, I omit that here.

Although you don't usually need to call it directly, `UnicastRemoteObject` has a corresponding `unexportObject` that forcibly removes an object from the RMI runtime. After you invoke this, if it succeeds, the object you pass can no longer accept incoming RMI calls.

Generating the Stub Classes for Java SE

If you'd like, you can generate Java SE stub classes for inclusion in Java SE applications; this can be handy if you want to deploy your application for both Java ME and Java SE, or if your distributed architecture calls for objects to be served from the Java ME device *to* hardware running Java SE. Interestingly, you don't perform this step on your Java source code; you perform it on the class files generated from your implementation.

For a small project like this, it's simply a matter of compiling `LocationImpl` and then running the RMI compiler `rmic` on the resulting class file. To do this, bring up a shell and issue the commands shown in Listing 11-3.

Listing 11-3. *Building the Java SE Stub Classes by Hand*

```
c:\book\Chapters\chapter11\code>javac Location.java
c:\book\Chapters\chapter11\code>javac LocationImpl.java
c:\book\Chapters\chapter11\code>rmic LocationImpl
```

These commands use `javac` to generate the class files for `Location.java` and `LocationImpl.java`; the final line uses `rmic` to generate the skeleton `LocationImpl_Stub.class`.

■ **Note** While the appearance of the command line differs if you're using Mac OS X or Linux, the command syntax remains the same.

Writing the Remote Service Host Application

The remote service needs to do two things. First, it must create an instance of implementation of the objects it serves. Second, it must register those objects with the RMI naming service, specifying where clients can gain access to the remote object. Listing 11-4 shows how to do this.

Listing 11-4. *A Simple RMI Server Application*

```
import java.rmi.Naming;

public class LocationServer {
   public LocationServer() {
       try {
           Location c = new LocationImpl();
           Naming.rebind("rmi://localhost:1099/LocationService", c);
       } catch (Exception e) {
           System.out.println("Exception: " + e);
       }
   }

   public static void main(String args[]) {
       new LocationServer();
   }
}
```

This is pretty simple stuff, and it demonstrates the power of building a remote application using RMI. The server creates a new instance of `LocationImpl` and registers it with the naming daemon—that's it!

To run the service on a stand-alone workstation for testing, you must first launch the RMI registry, which is the Java application with which the `LocationServer` registers its object for clients to find. You can launch `rmiregistry` in one shell and `LocationServer` (using `java` to start the Java VM) in the other.

■ **Tip** If you can't run the service, or if the client and server aren't communicating, check to see if you're running firewall or other software that would prevent `rmiregistry` from listening on the default port (1099). Another possibility is to operate `rmiregistry` on a different port, such as 8081, and change the port number in the application.

Of course, your production-caliber server isn't going to be running on your laptop with `rmiregistry`; instead you'll host the resulting implementation on an application server such as Sun's GlassFish or BEA's WebLogic Server. How you package and install the remote service differs slightly depending on the application server you choose.

Invoking the Remote Object from the Client

Invoking the remote object from the client is trivial: simply look up the remote service and obtain an instance of the remote object, as shown in Listing 11-5.

Listing 11-5. *Invoking the Remote Object*

```
import java.rmi.*;
…
        Location c = (Location)Naming.lookup(
                "rmi://localhost/LocationService");
…
```

When invoking a remote object, you must be sure that you specify the remote host—in this case, `localhost`—as well as the port and the name of the remote service providing the object. The `java.rmi.Naming` class does the rest, contacting the remote service and obtaining an instance of the remote object.

Wrapping Up

Some devices running the CDC support RMI OP, an optional extension that brings much of Java RMI to constrained devices. Like the Java SE implementation, RMI OP provides facilities for remote object discovery, remote communication, and remote behavior definition, as well as a secure communications channel that the Java implementation of RMI OP provides. Java RMI heavily leverages the use of interfaces to decouple an object's described behavior from its implementation. Remote clients use abstract interfaces describing how a remote object will behave in conjunction with an underlying object generated by the Java runtime that communicates with a remote server providing the object's actual implementation.

The RMI OP package, defined by JSR 66, provides a comprehensive subset of Java RMI, including remote call semantics, remote object representation, object exporting from CDC devices, and distributed garbage collection. RMI OP is based on the original J2SE RMI implementation and its wireline protocol; it is *not* an implementation of RMI-IIOP, which brings CORBA to Java. When writing applications that use RMI OP, you must define interfaces for the remote services, implement the remote services using the interfaces you define, and write the remote service host container that manages those interfaces. If part of your application runs on Java SE– or Java EE–enabled hardware, you must also generate the stub classes used by RMI to support object distribution across multiple virtual machines on the network.

Using Java RMI in Java ME isn't for everyone. In today's world of distributed web services using XML over HTTP, it's best suited for environments that require fully object-oriented approaches toward distributed computing, or for those legacy environments that require you to connect Java ME devices to existing network services already offering RMI services to other Java clients.

Intermezzo

This next part of the book is crucially important for you as a mobile applications developer. Today's Java ME platform is more connected than ever before, with required support for accessing today's Internet with the ubiquitous HTTP, as well as optional interfaces for lower-level Internet protocols as well as wireless protocols such as Bluetooth and SMS. Nearly every application of value requires at least some data communications capability. Plan on reading Chapter 12 closely, because its explanation of the GCF underpins communication within Java ME. If you're developing applications that utilize web services, read Chapter 13, because it teaches you how to work with Extensible Markup Language (XML), the lingua franca of the Internet. Wireless messaging (the topic of Chapter 14) is optional, but even if you aren't planning on using the wireless messaging interface, I recommend you at least skim this chapter so you have an understanding of the wireless communications options available to you.

PART 4

Communicating with the Rest of the World

Support for networked communications has been a crucial component of Java since the beginning, and Java ME continues this trend, bringing with it essential contributions to the entire community of Java developers. The three chapters in this part look at the key aspects of creating Java ME applications that use the Internet. In Chapter 12, you'll learn how to use the GCF, one of the most important frameworks and APIs provided by Java ME. It pervades Java ME's notions of how applications communicate across physical interfaces such as networks. In Chapter 13, you'll learn what options you have when working with XML—an important facet of many communicating applications today. Finally, in Chapter 14, you'll see how to access the wide-area wireless networking capabilities found on many Java ME devices.

CHAPTER 12

Accessing Remote Data on the Network

Sun Microsystems' early success was as a manufacturer of high-performance computing platforms for scientific and engineering purposes. A key feature of most workstations—including Sun's hardware—was support for networking. Embodied in the motto associated with Sun, "The Network Is the Computer," this commitment to networking continues to this day with the robust network solutions available to and crafted using the Java platform. Java ME is no exception; it was designed from the ground up as a platform that interacts with other computers on the network.

In this chapter, I talk about what is perhaps one of the most important contributions Java ME has made to the Java platform: the Generic Connection Framework (GCF). This framework provides a unified API for interacting with systems on the network using disparate protocols, and it has been extended to encompass file operations (see Chapter 7), wireless communications using SMS (see Chapter 14), and contactless communications (see Chapter 15). First, I explain what the GCF is and what it provides to you as an application developer. Next, I show you how to use the GCF to perform low-level communications on today's IP network using TCP and UDP. The bulk of the chapter, however, shows you how to use the GCF to access remote resources on the Internet using HTTP, an important facet of nearly every Java ME networked application. I close with a word on how the Java ME privilege model may affect your application.

Introducing the Generic Connection Framework

I/O and communication in the Java SE world is a wild ride: you find sockets, files, and connection classes (derived from `URLConnection`) residing in the `java.net` and `java.io` packages. Arguably powerful, to its detriment the Java SE framework for I/O requires you to have a command of a relatively large number of interfaces, many with dissimilar semantics. This turns out to be equally unwieldy for mobile devices as well; these deep and sophisticated class hierarchies don't scale well to mobile devices, especially those running the first releases of Java ME. Consequently, with the introduction of the first CLDC, Sun introduced the GCF, a straightforward hierarchy of interfaces and classes for managing connections

and performing I/O. The GCF has proven to be so popular that it has been widely adopted outside its initial application; platform architects now abstract communications to files, other network protocols, smart cards, radio-frequency identification (RFID) cards, Bluetooth peripherals, and even bar codes using the GCF. In fact, there's work underway to fold the GCF back into Java SE, as documented in JSR 197, demonstrating how the convergence of fixed and mobile computing paradigms draws from both environments.

Contained within the `javax.microedition.io` package, the GCF provides an abstracted approach to connectivity. By providing an extensible interface hierarchy of common classes (see Figure 12-1) and a factory to create instances of those classes based on a common request scheme based on URLs, the GCF significantly decreases the number of classes required to support communicating applications. This reduces both the Java ME platform footprint and the complexity that you as a Java ME developer must manage when building your application.

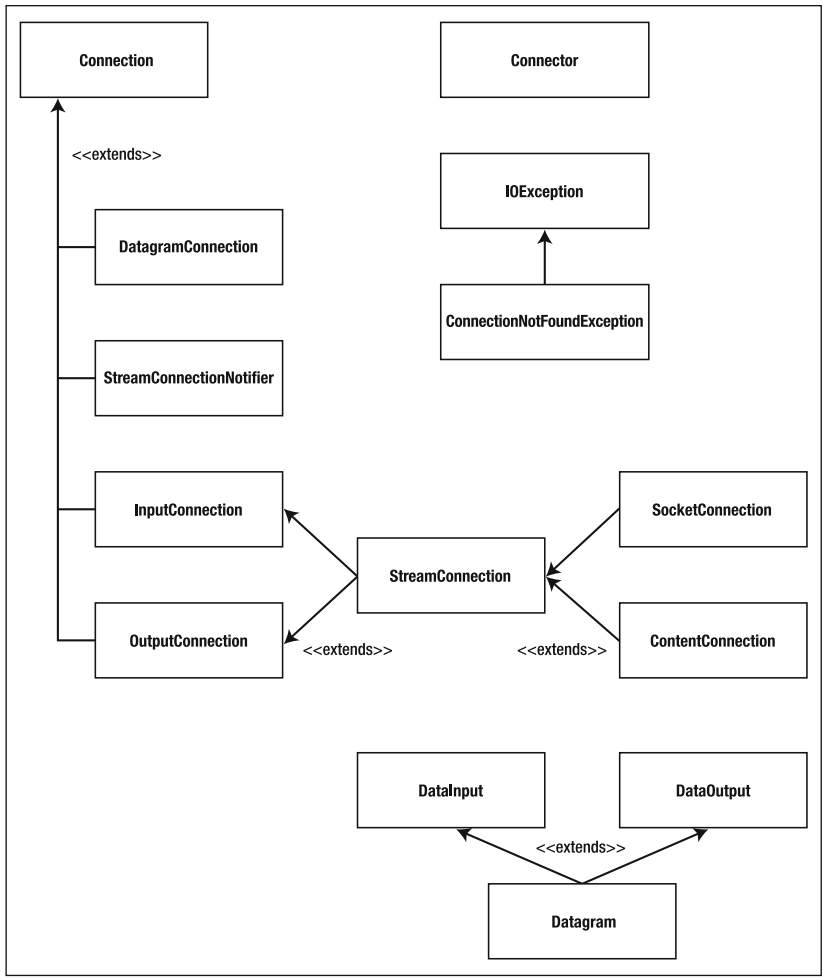


Figure 12-1. *The GCF hierarchy as originally introduced in CLDC 1.0*

The GCF hierarchy is a tightly woven collection of interfaces with a sprinkling of classes:

- *The* `Connector` *class*: Responsible for operating as a factory of `Connection` objects.
- *The* `Connection` *interface*: Represents a generic connection between two entities. The `Connection` interface defines the `close` method, which is responsible for closing a connection.
- *The* `DatagramConnection` *interface*: Defines the actions you can take on a datagram-oriented connection such as UDP.
- *The* `StreamConnection` *interface*: Defines the actions you can take on a stream-oriented connection such as TCP. `StreamConnection` instances are bidirectional, consisting of an `InputConnection` (from which you read data) and an `OutputConnection` (to which you write data).
- *The* `StreamConnectionNotifier` *interface*: Defines the actions you can take when listening for a new `StreamConnection`.
- *The* `ContentConnection` *interface*: Supports the passing of content encoded through a well-known codec.
- *The* `Datagram` *interface*: Defines an abstract interface for a single datagram. `Datagrams` are bidirectional, implementing this functionality by extending the `DataInput` and `DataOutput` interfaces.
- *The* `IOException` *class*: Signals generic communications errors such as network failures, and the `ConnectionNotFoundException`, which is thrown by `Connector` class methods when you attempt to create a connection that the platform cannot resolve. (This `IOException` class isn't formally part of the GCF.)

When you use the GCF, you follow these steps:

1. Create a URL describing the kind of socket you'd like to open and detailing the address and name of the resource to which you want to connect.
2. Invoke the `Connector.open` class method, passing the URL you constructed to receive an instance of a specific `Connection` subclass.
3. Use the concrete `Connection` subclass methods for the kind of connection `Connector.open` returned to you to work with the connection.

The pseudocode in Listing 12-1 demonstrates this pattern, opening an HTTP connection to the server `www.nowhere.com` and obtaining an `InputStream` from which to read.

Listing 12-1. *Opening an HTTP Connection and Obtaining an InputStream*

```
String url = "http://www.nowhere.com";
…
StreamConnection c = null;
InputStream s = null;
try {
    c = (StreamConnection)Connector.open(url);
    s = c.openInputStream();
    …
} catch (ConnectionNotFoundException e) {...}
  catch (IllegalArgumentException e) {...}
  catch (IOException ioe) {...}
finally {
    try {
        if (s != null) s.close();
        if (c != null) c.close();
    } catch (Exception e) {...}
}
```

This pseudocode is fairly typical for most applications that use connections. You begin by constructing a URL describing where the connection should be made, and then you use `Connection.open` to generate a concrete `Connection` class that implements the connection to the address you specified in the URL. Once the connection is open, you obtain streams (or datagrams) from the connection and perform your I/O using the provided streams or datagrams. When you're finished, you need to close both the stream (not the datagram connections) and the connection itself.

Programming with connections is fraught with exceptions, because you're at the mercy of the outside environment. A network might not be available, a handset might not have service, or sufficient resources may not be available on the device to accommodate your network request. Consequently, `open` can throw a number of exceptions:

- `ConnectionNotFoundException`: If the platform doesn't support the requested protocol, or if the platform cannot find the target specified by the URL
- `IllegalArgumentException`: If you specify an invalid parameter
- `IOException`: If any kind of I/O error occurs while opening the connection
- `SecurityException`: If your application is not permitted to access the protocol you request

While the example in Listing 12-1 passes only the URL to `open`, you can also specify an access mode (one of `Connector.READ`, `Connector.WRITE`, or `Connector.READ_WRITE`), indicating the access mode for the resulting connection. If you supply an access mode, you can also indicate that your application wants to receive time-outs when using the connection if they're provided by the specific protocol.

You're probably familiar with the notion of a URL, but because URLs play a key role in the creation of connections, it's worth reviewing their semantics. Using the GCF, URLs describe both the kind of connection as well as the location to which the connection instance should connect. The syntax of a URL is described in RFCs 1738 and 2396; a URL is a single string that looks like this:

```
scheme://user:password@host:port/path;parameters
```

where

- The `scheme` specifies the protocol used for the connection, such as HTTP.
- The `user` specifies an optional username required when accessing the connection.
- The `password` specifies an optional password required when accessing the connection.
- The `host` is the fully qualified domain name or other address of the remote end of the connection.
- The `port` specifies an optional port to be used on the remote end of the connection.
- The `path` is a path to the remote end of the connection, whose interpretation and format may vary depending on the `scheme`. The path may include one or more parameters modifying the `connection`.

Under the hood, the implementation of `Connector` uses the scheme you specify in a URL to determine the kind of `Connection` it should instantiate and return. Java ME and the various RFCs for extending the communication capabilities of Java ME have defined several different schemes. Table 12-1 provides a list of common schemes, the protocol, the returned `Connection` subclass, and in which JSR the scheme was introduced to the Java ME platform.

Table 12-1. *Connection Schemes*

URL Scheme	Protocol	GCF Connection Type	Defined By	Required or Optional
`btl2cap`	Bluetooth	`L2CAPConnection`	JSR 82	Optional
`datagram`	Datagram	`DatagramConnection`	CLDC, CDC, and JSR 197 for J2SE	Optional
`file`	File system access	`FileConnection`, `InputConnection`	JSR 75†	Optional
`http`	HTTP	`HttpConnection`	MIDP 1.0,* MIDP 2.0,** Foundation Profile,*** and JSR 197 for Java SE	Required
`https`	Secure HTTP	`HttpsConnection`	MIDP 2.0**	Required
`comm`	Serial I/O	`CommConnection`	MIDP 2.0**	Optional
`sms`	SMS	`MessageConnection`	JSR 120, JSR 205	Optional
`mms`	MMS	`MessageConnection`	JSR120, JSR 205	Optional
`cbs`	Cell Broadcast Service (CBS)	`MessageConnection`	JSR 120, JSR 205	Optional
`apdu`	Application Protocol Data Unit (APDU)	`APDUConnection`	JSR 177††	Optional
`jcrmi`	Java Card Remote Method Invocation	`JavaCardRMIConnection`	JSR 177††	Optional
`socket`	Socket	`SocketConnection`	MIDP 2.0**	Optional
`serversocket`	Socket	`ServerSocketConnection`	MIDP 2.0**	Optional
`datagram`	UDP	`UDPDatagramConnection`	MIDP 2.0**	Optional

**JSR 37 defines MIDP 1.0.*

***JSR 118 defines MIDP 2.0.*

****JSR 46 defines Foundation Profile 1.0; JSR 219 defines Foundation Profile 1.1.*

†I discuss this in Chapter 7.

††I discuss this in Chapter 15.

From Table 12-1, you can see that not all Java ME platforms support even the most common connection types. This is something you must keep in mind when designing your applications. For example, it would do you little good to develop an application that required the use of UDP datagrams for widespread deployment on legacy handsets that probably run CLDC MIDP 1.0, because those handsets likely don't provide support for the `UDPDatagramConnection`, and even some CLDC MIDP 2.0 handsets lack support for that class. In general, it's safe to assume support for HTTP on any CLDC or CDC device;

secure connections through HTTPS are usually available on MIDP 2.0 implementations, but some devices may lack HTTPS support. Raw (stream or datagram) socket support is less common, and serial support is significantly less common than even stream or datagram support. Other schemes, like those supporting SMS (which I describe in Chapter 14), Bluetooth, and the FCOP (see Chapter 7), vary in their availability depending on the carrier and handset class you target. It's worth your effort to do some market research as you define the features for your application to ascertain how many devices implement the optional packages you need. Table 12-2 summarizes the availability of specific `Connection` implementations by profile.

Table 12-2. *Connection Implementations*

Connection	Required?	CLDC 1.0, 1.1	MIDP 1.0	MIDP 2.0	Foundation and Related Profiles
`CommConnection`	N			✓	
`Connection`	Y	✓	✓	✓	✓
`ContentConnection`	Y	✓	✓	✓	✓
`DatagramConnection`	N	✓	✓	✓	✓
`HttpConnection`	Y		✓	✓	✓
`HttpsConnection`	Y			✓	
`InputConnection`	Y	✓	✓	✓	✓
`OutputConnection`	Y	✓	✓	✓	✓
`SecureConnection`	N			✓	
`ServerSocketConnection`	N			✓	
`SocketConnection`	N			✓	
`StreamConnection`	Y	✓	✓	✓	✓
`StreamConnectionNotifier`	Y	✓	✓	✓	✓
`UDPDatagramConnection`	N			✓	

The GCF isn't perfect: critics may fairly claim that the `Connection` class doesn't go far enough in abstracting common communications operations into abstract methods, so that many conceptually similar operations (such as sending a datagram to a remote host or sending a protocol data unit to a smart card) have widely varying interface semantics in different subclasses of the `Connection` class. This is unfortunate, because it makes learning about the various communications options available through optional packages more difficult, and it makes taking a component-oriented approach to developing communicating applications impossible without the use of adapter classes.

Communicating with Sockets and Datagrams

Although the vast majority of networked mobile applications likely use HTTP to fulfill their communications needs, some applications may still need raw socket or datagram facilities. This includes utilities for system administrators, instant messaging clients, and novel applications of these protocols to games or multimedia applications.

Unfortunately, support for these facilities—usually implemented over TCP for sockets and UDP for datagrams—is not available on all devices. As Table 12-2 shows, MIDP 2.0 optionally provides support for socket communication, while all Java ME platforms may provide support for datagram communication. Consequently, as you establish the business case for your application, you should consider the availability of these protocols and do some research to determine precisely which devices will support your application, and if there are enough devices in the hands of your target market to justify your development expense. If not, you may want to consider tunneling your data communications over another protocol such as HTTP.

■ **Tip** Resources such as the Java ME Device Table at `http://developers.sun.com/mobility/device/device` and the WURFL device description repository at `http://wurfl.sourceforge.net` can help you determine which devices support socket or datagram communications.

Using Sockets with the GCF

Socket programming with the GCF is easy: construct a URL to the destination of the socket, and open it using `Connector`. Assuming everything goes your way—the path to the server is available and the server is running—you can then obtain streams to perform input and output on the connection established by the device.

When opening a socket connection, `Connector.open` returns an instance of `SocketConnection`, which implements `StreamConnection`. For *most* socket-oriented programming, using a `StreamConnection` is sufficient, as shown in Listing 12-2.

Listing 12-2. *Using a StreamConnection*

```
String url = "socket://nowhere.com:7";
…
StreamConnection c = null;
InputStream s = null;
OutputStream o = null;
byte[12] b;
```

```
try {
    c = (StreamConnection)Connector.open(url);
    s = c.openInputStream();
    o = c.openOutputStream();
    o.write("Hello world!".getBytes());
    s.read( b, 0, b.length );
} catch (ConnectionNotFoundException e) {...}
  catch (IllegalArgumentException e) {...}
  catch (IOException ioe) {...}
finally {
    try {
        if (o != null) o.close();
        if (s != null) s.close();
        if (c != null) c.close();
    } catch (Exception e) {...}
}
```

This code opens a TCP socket to port 7—the echo port—and proceeds to write the message "Hello world!" Once it writes the message, it reads the response and then tears down the streams and connection. As you see from the code, you perform the actual I/O using the input and output stream classes `InputStream` and `OutputStream`; they let you exchange individual arrays of bytes using their `read` and `write` methods. For higher-level access, you can also obtain `DataInputStream` and `DataOutputStream` instances from the connection using the `StreamConnection`'s methods `openDataInputStream` and `openDataOutputStream`; as you recall from Chapter 2, they let you read and write primitive data types including floating-point numbers, integers, and strings.

The `SocketConnection` instance that `Connector.open` returns has a few more methods than the `StreamConnection`, and it lets you perform some low-level operations on the socket by invoking the following methods:

- `getAddress`: Lets you obtain the remote address to which the socket is bound by invoking on the socket
- `getLocalAddress`: Lets you obtain the local address to which the socket is bound
- `getPort`: Lets you obtain the remote port to which the socket is bound
- `getLocalPort`: Lets you obtain the local port to which the socket is bound
- `getSocketOption`: Lets you get a socket option
- `setSocketOption`: Lets you set a socket option

The socket options are a subset of the socket options you may have encountered using TCP sockets on Linux or UNIX; Table 12-3 summarizes the options you can obtain or mutate. You pass the option identifier to `getSocketOption`, and it returns the resulting option; you pass the option identifier and a new value to `setSocketOption`, and it attempts to set the option you specify.

Table 12-3. *Socket Options*

Option Identifier	Meaning
`SocketConnection.DELAY`	Duration to delay when writing to small buffers
`SocketConnection.KEEPALIVE`	Duration that a socket should remain open for keep-alive purposes
`SocketConnection.LINGER`	Duration that a socket should linger open with output remaining when there's no response from the other side of the socket
`SocketConnection.RCVBUF`	Size in bytes of the receiving buffer
`SocketConnection.SNDBUF`	Size in bytes of the sending buffer

The GCF also has support for accepting incoming socket requests through the `StreamConnectionNotifier` and `ServerSocketConnection` classes, although in practice Java ME applications rarely use this facility. There are at least three reasons why you probably don't want a Java ME application listening on a socket for incoming connections. First, listening for an incoming connection requires the hardware to supply power to the network subsystem. For many portable devices, especially those with wireless radios, this can drastically reduce battery life. Second, the networks on which most Java ME devices operate typically use dynamic address assignment when allocating addresses to Java ME devices; even if there weren't energy penalties for operating the network hardware, you wouldn't know what address corresponded to a specific device without using some kind of registration mechanism in advance. Finally, only a small number of devices offer this support.

If you're sure these reasons don't apply to you, you can obtain a `ServerSocketConnection` by simply specifying the port on which you want to listen, as shown in Listing 12-3.

Listing 12-3. *Obtaining a ServerSocketConnection*

```
String url = "socket://7";
ServerSocketConnection ssc = null;
try {
    ssc = (ServerSocketConnection)Connector.open(url);
    while (true) {
        SocketConnection sc = (SocketConnection)ssc.acceptAndOpen();
        InputStream s = sc.openInputStream();
        OutputStream o = sc.openOutputStream();
```

```
        try {
            ...
        }
        catch(Exception e) {...}
        finally {
           if (s!=null) s.close();
           if (o!=null) o.close();
           if (sc!=null) sc.close();
        }
    }
} catch (Exception e) {...}
finally {
    try {
        if (ssc != null) ssc.close();
    } catch (Exception e) {...}
}
```

This code strongly resembles that of a traditional select/accept server loop written using UNIX sockets. After creating a `ServerSocketConnection`, the code blocks on the `ServerSocketConnection` instance's `acceptAndOpen` method, which returns a `SocketConnection` instance once a client has connected. Next, it opens input and output streams using the socket to perform the necessary I/O service; once this is done, it cleans up and goes back to waiting for the next request.

Of course, the accept/respond loop only runs for the lifetime of your application; a true server would likely need to listen even when your application is not running. On MIDP platforms, you can accomplish this using the push registry by registering your application for incoming requests either at installation time or at runtime. To register at installation time, simply include the GCF URL describing the socket to which you want to listen and the name of your MIDlet as a value for a `MIDlet-Push` field in the application descriptor, like this:

```
MIDlet-Push-1: socket://:7, ServerMIDlet, *
```

This instructs the application management system to invoke your application whenever an incoming TCP request appears on port 7. Your application can then use the `PushRegistry`'s `listConnections` method—which I describe in Chapter 14—to obtain a list of connections that have data waiting. You can also programmatically add your application to the push registry by invoking its `registerConnection` method, passing exactly the same information as you provide in the `MIDlet-Push` field of your application descriptor.

Using Datagrams with the GCF

Datagram communication—which is generally suited for short, single messages where sequencing and flow control are unnecessary—uses the `Datagram` class to encapsulate the concept of a datagram. Like sockets, you use the `Connector` class to open a connection; when using datagram communication, you obtain a class such as `DatagramConnection` or `UDPDatagramConnection`, with which you create an instance of `Datagram`. Once you have the `Datagram` instance in hand, you use it to send and receive individual datagrams. Listing 12-4 writes "Hello world!" to a datagram echo server.

Listing 12-4. *Writing "Hello world!" to a Datagram Echo Server*

```
String url = "datagram://noplace.com:7";
DatagramConnection dgc = null;
Datagram d;
String s = "Hello world!";
byte m[] = s.getBytes();
byte[] b = new Byte[12];

try {
    dgc = (DatagramConnection)Connector.open(url);
  try {
      d = dgc.newDatagram(m, m.length);
      dgc.send(d);
      d.reset();
      dgc.receive(d);
      d.readFully(b, 0, b.length);
  } finally {
      dgc.close();
  }
} catch (Exception e) {...}
finally {
    try {
        if (dgc != null) dgc.close();
    } catch (Exception e) {}
}
```

The GCF pattern here is clear, although instead of using streams for input and output, you use the `Datagram` class. The `DatagramConnection` class acts as a factory for `Datagram` objects, which you then pass to `send` and `receive` to send and receive single datagrams, respectively.

The `Datagram` class is an abstract class that encapsulates the following concepts common to all datagram protocols:

- Datagrams have a *buffer* in which the datagram stores the data in a single datagram packet.
- Datagrams have an *offset*, which is the current position for reading or writing data in the buffer.
- A datagram's buffer has a *length*, indicating the size of the datagram buffer.
- Datagrams have a representation of the source or destination address of the datagram.

Datagrams implement the `DataInput` and `DataOutput` interfaces, as you saw in Figure 12-1, letting you read and write various primitive data types, including `boolean`, `byte`, `character`, `double`, `float`, `integer`, `long integer`, and UTF `Strings`. The interface to `DataInput` and `DataOutput` is reminiscent of that provided by `DataInputStream` and `DataOutputStream`; the classes provide methods `read`*X* and `write`*X* for the various supported types *X*, including the following:

- `readBoolean`, which lets you read an input byte and returns `true` if the byte is nonzero, and `writeBoolean`, which lets you write a `boolean` as a single byte
- `readByte` and `writeByte`, which let you read and write single bytes, respectively
- `readChar` and `writeChar`, which let you read and write single characters, respectively
- `readFloat` and `writeFloat`, which let you read and write floating-point numbers, respectively
- `readShort` and `writeShort`, which let you read and write short integers, respectively
- `readInt` and `writeInt`, which let you read and write integers, respectively
- `readLong` and `writeLong`, which let you read and write long integers, respectively
- `readUnsignedByte` and `writeUnsignedByte`, which let you read and write a single unsigned byte, respectively
- `readUTF` and `writeUTF`, which let you read and write UTF-8 encoded strings, respectively
- `readFully` and `write`, which let you read and write arrays of bytes, respectively

Each `Datagram` instance represents a single message either waiting to be written to the connection or previously read from the connection; you can reuse a `Datagram` instance by invoking its `reset` method. You can also query the various properties of a `Datagram` using the following methods:

- `getAddress`: Returns the address of a `Datagram`
- `getData`: Returns the contents of a `Datagram`'s buffer
- `getOffset`: Returns the offset into a `Datagram`'s buffer
- `setData`: Lets you set a `Datagram`'s buffer, offset, and length given an array of bytes
- `setLength`: Lets you set the length of a `Datagram`'s buffer

Communicating with HTTP

After the protocols that deliver e-mail, HTTP is perhaps the most widely deployed and understood protocol for data exchange today. At heart, it's a simple client/server protocol, and its simplicity and generality have led it to be one of the most important protocols on the Internet today. Besides being the protocol responsible for delivering content on the Web—created by Tim Berners-Lee, it has its origins in providing the transport for hypertext documents at the European Organization for Nuclear Research (CERN)—it now underpins many other client/server exchanges, such as those enabling web services. (I talk more about the web service model of computing in the next chapter.)

Reviewing HTTP

HTTP is a client/server protocol in which a client application—the *user agent*—makes a *request* of a server for an operation or content located at a particular URL. The request consists of *headers* that contain metadata about the request, and an *object body*, which is an optional block of data that pertains to the request. The server replies to the request in the same way, returning headers that provide metadata about the request followed by an optional object body in response. HTTP can run over any reliable stream protocol, but in common practice, it operates over TCP. The protocol itself uses an eight-bit representation for characters, but the request and response headers are written as plain text, making it easy for developers to understand, implement, and troubleshoot. For example, a client web browser might send the message in Listing 12-5 to obtain the home page at the URL `http://www.noplace.com`.

Listing 12-5. *Obtaining a Web Page*

```
GET /index.html HTTP/1.1
Host: www.noplace.com
```

The first line of the request is the request itself; the request consists of a method, a resource, and the version of the protocol. The `GET` request asks the remote server to return the indicated resource—in this case, the document `index.html`—using HTTP version 1.1. Subsequent lines of the request consist of metadata about the request, indicated as name-value pairs demarcated by colons and separated by newlines, such as the server destination host (in the example, `www.noplace.com`). If a request includes an object body, it follows the headers after a blank line, as shown in Listing 12-6.

Listing 12-6. *Posting Information to a Web Server*

```
POST /do HTTP/1.1
Host: www.noplace.com

name=value
```

Here, the remote server is being asked to process the object body `name=value` by invoking the script `do` and passing the object body to the script. Table 12-4 shows a list of the defined HTTP methods (also called *verbs*) and their use.

Table 12-4. *HTTP Methods and Their Meaning*

Method	Meaning
`HEAD`	Asks for the response identical to the response created by a `GET`, but without the object body
`GET`	Requests a representation of the specified resource
`POST`	Submits data to be processed by the specified resource
`PUT`	Uploads a representation of the specified resource
`DELETE`	Deletes the specified resource
`TRACE`	Echoes back the received request
`OPTIONS`	Returns the raw HTTP methods that the server supports for the resource
`CONNECT`	Converts the request connection to a transparent TCP tunnel

Returning to Listing 12-5, the server might make a response like the one shown in Listing 12-7.

Listing 12-7. *A Typical Server Response*

```
HTTP/1.1 200 OK
Date: Mon, 23 Jun 2008 22:13:40 GMT
Server: Apache/2.2.8 (Unix) mod_ssl/2.2.8 OpenSSL/0.9.7l DAV/2 PHP/5.2.5
Content-Location: index.html
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Mon, 24 Sept 2007 01:12:03 GMT
ETag: "ec11-5b0-43ad74ee73ec0;44bbb82e73280"
Accept-Ranges: bytes
Content-Length: 49
Connection: close
Content-Type: text/html

<html>
  <body>
  Hello world!
  </body>
</html>
```

The first line of an HTTP server's response is always the response status, consisting of the server version, a numeric indication of status, and a human-readable status string. A status code is always a three-digit number; the first digit indicates the class of the response, and the second and third indicate details. Common classes are `200`, indicating success, and `400`, indicating a client error that caused the server to be unable to fulfill the request. Table 12-5 shows a list of common (but not all) HTTP status codes. Following the response status are a series of headers; the actual meaning of each header isn't as important now as understanding that each header has a name (such as `Date`) followed by a value (`Mon, 23 Jun 2008 22:13:40 GMT`). You may encounter some headers such as `Content-Length` and `Content-Type` often when using HTTP; many others may not be necessary for your work. Fortunately, the most common ones, such as the `Date`, `Content-Length`, and `Content-Type` headers, are self-explanatory; they indicate the date and time the server generated the response, how many bytes are in the response, and the encoding method the server used to encode the response, respectively. Finally, after the headers comes the object body of the response—in this case, a short HTML document.

Table 12-5. *Common HTTP Status Codes*

Status Code	Meaning
100	Request headers have been received; the client may send the request body.
101	Switching protocols.
200	Standard response for successful HTTP requests.
201	The request has been completed and has resulted in a new resource.
202	The request has been accepted for processing, but the processing has not been completed.
204	No content.
301	Moved permanently; this and all future requests should be directed to the given URL.
302	Moved temporarily; this request should be directed to the given URL.
303	The response can be found under another URL using a `GET` method request.
304	The resource has not been modified since last requested.
400	The request contains bad syntax or cannot be completed.
401	Authentication of the user agent is possible but did not occur.
403	The request was legal, but the server is refusing to respond to it.
404	The requested resource could not be found.
405	A request was made of a resource using a method not supported by that resource.
500	An internal server error occurred while handling the request.
503	The service is presently unavailable.
505	The HTTP version used is not supported.
509	The bandwidth limit was exceeded.*

**This is not an official status code, but is returned by many servers.*

Using HTTP with the GCF

HTTP's ability to provide metadata with requests for service using standard verbs lets you define a wide variety of services that a remote system can provide besides just returning pieces of a web page. Often, though, Java ME devices need only the ability to read remote content. You can download content using the GCF and a `ContentConnection`, as shown in Listing 12-8.

Listing 12-8. *Using a ContentConnection to Download an Image*

```
public void run() {
    ContentConnection cc = null;
    DataInputStream in = null;
    try {
        String url = "http://www.noplace.com/image.png";
        cc = (ContentConnection)Connector.open(url);
        in = new DataInputStream(cc.openInputStream());
        int length = (int)cc.getLength();
        byte[] data = null;
        if (length != -1) {
            data = new byte[length];
            in.readFully(data);
        }
        else {
            int chunkSize = 512;
            int index = 0;
            int readLength = 0;
            data = new byte[chunkSize];
            do {
              if (data.length < index + chunkSize) {
                  byte[] newData = new byte[index + chunkSize];
                  System.arraycopy(data, 0, newData, 0, data.length);
                  data = newData;
              }
              readLength = in.read(data, index, chunkSize);
              index += readLength;
          } while (readLength == chunkSize);
          length = index;
        }
        Image image = Image.createImage(data, 0, length);
        …
    }
    catch (Exception e) {...}
    finally {
        try {
            if (in != null) in.close();
        }
        catch (IOException ioe) {}
    }
}
```

The `ContentConnection` class inherits from `StreamConnection` and provides three new methods: `getEncoding`, `getLength`, and `getType`. This example uses the `getLength` method to determine the number of bytes being returned by the remote server; unfortunately, there's no guarantee that the server will actually report this information. Consequently, this example tests the returned content length; if it's undefined, the code will read the data in 512-byte chunks. Either way, the code creates a new image with the returned data and closes both the input stream and `ContentConnection`. This code is encapsulated in a runnable method, `run`, because in practice you're likely to use threads for all of your I/O. (I show you a concrete example of this in the "Putting HTTP to Work" section, which details a full application that uses HTTP.)

The `ContentConnection` class is a high-level abstraction of the kind of metadata that HTTP provides, and it lets you perform only a fetch of remote content. Using the `ContentConnection`, you specify a remote resource when you create the connection using the `Connector` class. Under the hood, the platform makes the desired request using the HTTP method `GET` and the connection you specify when you request the `DataInputStream` from which the remote resource will be read, giving you no control over the semantics of the HTTP request itself. Once the platform has completed the request, you can learn only three things about the response: how the response was encoded (using the `getEncoding` method), the number of bytes returned (using the `getLength` method), and the Multipurpose Internet Mail Extensions (MIME) type of the method (using the `getType` method).

■ **Note** The content type and other information about the response are only available if the server provides them; your application must be prepared to deal with the possibility that a server won't return this information (unless you have control over the server implementation as well, and even then you would need to handle any errors that arise in your application from a lack of this metadata).

Because `ContentConnection` is such a simple interface, it's ideal when your application needs only to obtain the contents of a remote resource. It is, however, woefully inadequate for more sophisticated uses of HTTP, because it does not provide access to most of the metadata that accompanies HTTP requests and responses, nor does it let you specify the method HTTP invokes when making the request. For finer-grained access to the HTTP protocol, you need to use instances of the `HttpConnection` class—a class that inherits from `ContentConnection`. (In fact, when you invoke `Connector.open` on a URL with the `http` scheme, the instance returned is the `HttpConnection`, not the `ContentConnection` the code in Listing 12-8 suggests.)

Reflecting the underlying connection with a remote HTTP server, an `HttpConnection` instance can be in one of three states:

- `Setup`: You can set request parameters, including the method.
- `Connected`: The request method and parameters have been sent, and the response is requested.
- `Closed`: The HTTP connection has been terminated.

Unlike other types of connections, you need to know what state an `HttpConnection` is in, because certain methods will only work when the connection is in the proper state. For example, you can't set the method type the connection should use once the connection is in the `connected` state, because the connection has already sent the request. The transition from `setup` to `connected` occurs when you invoke any method that requires the connection to exchange data, such as opening a stream for reading or writing to the connection.

While the connection is in the setup state, you can set the request method using the connection's `setRequestMethod` method, passing one of `HttpConnection.GET`, `HttpConnection.POST`, or `HttpConnection.HEAD`. You can also specify an arbitrary request header for the request, such as the MIME content type being provided. You do this using the connection's `setRequestProperty` method, passing the name of the request header and the value as strings, like this:

```
hc.setRequestProperty("Content-Type", "application/x-urlformencoded");
```

Many of the `HttpConnection` methods have direct correspondence with HTTP response headers, including the following methods:

- `getDate`: Returns the value of the `Date` header as a time in seconds since the beginning of the epoch
- `getExpiration`: Returns when the resource will expire, as indicated by the `Expires` header as a time in seconds since the beginning of the epoch
- `getLastModified`: Returns when the resource was last modified, as indicated by the `Last-Modified` header as a time in seconds since the beginning of the epoch
- `getResponseCode`: Returns the three-digit HTTP response code
- `getResponseMessage`: Returns the HTTP response message, if any
- `getHeaderField`: Returns the value of a specific HTTP header by name

You can also enumerate all of the response headers using the `getHeaderField` and `getHeaderFieldKey` methods by passing an integer; they return the value and name of the *n*th header, treating the block of response headers as a single array.

The HttpConnection class also contains some of the utility functions for examining URLs. These utility functions let you determine the scheme, host name, port, and file a URL contained when the GCF created the HttpConnection instance. These HttpConnection methods include getProtocol, getHost, getPort, and getFile. You can also determine the URL to which a specific HttpConnection refers by invoking its getURL method.

Probably the most common reason for needing to use an HttpConnection instead of a ContentConnection is when you need to submit data for processing, like user input to a web service. This may involve using the HTTP POST method instead of GET. You use an HttpConnection and get its OutputStream in order to write the data to the server, as shown in Listing 12-9.

Listing 12-9. *Sending Data to the Server*

```
void run() throws IOException {
    String url = "http://www.noplace.com/do";
    HttpConnection hc = null;
    DataInputStream is = null;
    OutputStream os = null;
    int rc;

    try {
        hc = (HttpConnection)Connector.open(url);

        hc.setRequestMethod(HttpConnection.POST);
        hc.setRequestProperty("Content-Type",
            "application/x-urlformencoded");

        os = hc.openOutputStream();
        os.write("name=value\n".getBytes());

        rc = hc.getResponseCode();
        if (rc != HttpConnection.HTTP_OK) {
            throw new IOException("HTTP response code: " + rc);
        }

        is = hc.openInputStream();

        int length = (int)hc.getLength();
        byte[] data = null;
        if (length != -1) {
            data = new byte[length];
            is.readFully(data);
        }
```

```
        else {
            int chunkSize = 512;
            int index = 0;
            int readLength = 0;
            data = new byte[chunkSize];
            do {
              if (data.length < index + chunkSize) {
                  byte[] newData = new byte[index + chunkSize];
                  System.arraycopy(data, 0, newData, 0, data.length);
                  data = newData;
              }
              readLength = is.read(data, index, chunkSize);
              index += readLength;
            } while (readLength == chunkSize);
            length = index;
        }
        …
    } catch (Exception e) {…}
    finally {
        try {
            if (is != null) is.close();
            if (os != null) os.close();
            if (hc != null) hc.close();
        }
        catch (Exception e) {…}
    }
}
```

The setup using the GCF is exactly the same; the only difference is that this time the code casts the result to an `HttpConnection`. The code then sets the request method to `POST` using `setRequestMethod`, and it adds an additional header to the request describing the content type. On most implementations, the process of obtaining an output stream for the request will cause the `HttpConnection` to move from the `setup` to the `connected` state; the remote server is now waiting for the application to write the object body for the `POST` request. The code does this by first obtaining the output stream on the connection using `openOutputStream`, and then by writing a simple form-encoded name-value pair. Even if the implementation buffers the headers and object body you've prepared using the `HttpConnection` instance and `OutputStream`, the `HttpConnection` *must* open the connection, send the request, and read the HTTP response headers when you invoke `getResponseCode`. At this point, the connection is definitely in the `connected` state, and you can obtain information about the result using connection methods such as `getLength`, as this code does to determine the length of the resulting object body. The remainder of the code is the same as

for using a `ContentConnection` and `InputStream` to read the results from the server, closing the streams and connection when the code finishes reading from the stream.

Putting HTTP to Work

Using the GCF for connections to remote resources is simple in theory, but in practice it's a little more complex, for two reasons. First, and most importantly, you must handle all of the exceptions that may arise; the pseudocode you've seen before indicates that exceptions *can* occur, but the pseudocode doesn't do anything useful with them. Second, you want to perform your I/O on a separate thread, because the connections usually block the thread on which they operate while they exchange data with the remote server. While on some devices and networks, this delay can be negligible, there's no guarantee of this, and the result of having connections perform their work on the main thread is a frozen user interface—something intolerable for users.

Listing 12-10 shows the `WeatherFetcher` class, which uses HTTP to obtain the weather forecast for a specific city and state. The WeatherWidget example uses it to obtain weather forecasts from a remote web server, as you see in Listing 12-11. (If you're following along with the listings in this chapter, you will want to make the changes in *all* of the listings in your project, because they are interrelated.)

■ **Note** The implementation of `WeatherFetcher` assumes that the remote web server can provide weather forecasts given a city and state, and that the resulting weather forecasts are provided to the application in plain text. In the next chapter, I build on what you learn here to see how an actual web service would use XML to provide structure to the request and response data.

Listing 12-10. *The WeatherFetcher Class*

```
package com.apress.rischpater.weatherwidget;

import java.io.*;
import javax.microedition.io.*;

public class WeatherFetcher implements Runnable {
    private String url = "http://www.noplace.com/weather";
    private Thread thread;
    private Location location;
    private boolean cancelled;
    WeatherWidget app;
```

```
public WeatherFetcher(Location l, WeatherWidget a) {
    location = l;
    app = a;
    cancelled = false;
    if (l!=null && a!=null) {
        thread = new Thread(this);
        thread.start();
    }
}

public void cancel() {
    cancelled = true;
}

public void run() {
    String vars;
    String forecast = "";
    HttpConnection hc = null;
    InputStream in = null;
    OutputStream out = null;

    vars = "location="+WeatherFetcher.urlEncode(location.getLocation());

    try {
        hc = (HttpConnection)Connector.open(url);
        hc.setRequestMethod(HttpConnection.POST);
        hc.setRequestProperty("Content-Type",
            "application/x-www-formrlencoded");
        hc.setRequestProperty("Content-Length",
            Integer.toString(vars.length()));
        out=hc.openOutputStream();
        out.write(vars.getBytes());
        in=hc.openInputStream();
        int length=(int)hc.getLength();
        byte[] data = new byte[length];
        in.read(data);
        forecast = new String(data);
    }
```

```
            catch(Exception e){ forecast = e.getMessage(); }
            finally {
                try {
                    if (in!=null) hc.close();
                    if (out!=null) hc.close();
                    if (hc!=null) hc.close();
                }
                catch(Exception e) {}
            }
            if (!cancelled) {
                location.setForecast(forecast);
                app.update();
            }
        }

        private static String urlEncode(String s)
        {
            if (s!=null) {
                StringBuffer tmp = new StringBuffer();
                int i=0;
                try {
                    while (true) {
                        int b = (int)s.charAt(i++);
                        if ((b>=0x30 && b<=0x39) || /* 0-9 */
                            (b>=0x41 && b<=0x5A) || /* A-Z */
                            (b>=0x61 && b<=0x7A)) { /* a-z */
                            tmp.append((char)b);
                        } else {
                            tmp.append("%");
                            if (b <= 0xf) tmp.append("0");
                                tmp.append(Integer.toHexString(b));
                        }
                    }
                }
                catch (Exception e) {}
                return tmp.toString();
            }
            return null;
        }
    }
```

The `WeatherFetcher` class implements `Runnable`, because the actual connection occurs in a separate thread. This permits the MIDlet to perform the network communication without blocking the user interface—a feature expected by today's users. Given a location and an instance of the application to notify when the network operation is complete, the class creates a new thread, connects to the remote server, posts the location to the server, and obtains the results. This work is started in the class constructor, which sets aside the provided location and reference to the application's user interface before starting a new thread using the newly created instance of this class. The `WeatherFetcher` class requires a reference to the MIDlet that uses it; the class notifies the MIDlet when the update has occurred by invoking the MIDlet's `update` method. Listing 12-11 shows this change to the WeatherWidget MIDlet, and I'll discuss this change more in a bit.

The user interface can cancel the operation at any time; this can occur if you pick a different location while the `WeatherFetcher` and its thread are blocked awaiting a response from the network. To do this, the user interface invokes the `cancel` method; it simply asserts a cancellation flag, which the `WeatherFetcher` tests before updating the user interface. In fact, `cancel` doesn't actually cancel the HTTP connection, because there's really no good way to do this; instead, it permits the HTTP connection to complete and cancels updating the user interface.

The `run` method actually performs the work necessary to obtain a weather forecast for a specific location; this code should be familiar to you by now. The `WeatherConnection` class assumes the remote web server has accepted the desired location as a name-value pair encoded as a form argument; this is typical for simple web servers that also provide content for web forms. Consequently, the first thing that `run` does is create a name-value pair and store the result in `vars`; the syntax of this pair is appropriate for any web server responding to a form, and looks like this:

```
location=Berkeley,CA
```

This step must *URL-encode* the argument string; some characters such as a space character (`" "`) cannot be transmitted as they are, but instead are represented as a single percent symbol (`%`) followed by the two-digit hexadecimal ASCII code of the character. Thus, a space character (`" "`) would be represented as the string `%20`. While the example passes only a single named value `location`, it could just as easily pass multiple values; these would be concatenated using an ampersand (`&`), like this:

```
city=Berkeley&state=CA
```

Typically, however, if you're building an application to exchange structured data with a web server, you'll probably encode that structured data using XML—a subject I save for Chapter 13.

Tip If you're passing name-value pairs using the HTTP `GET` or `POST` methods, it's imperative that you URL-encode the values, or else the URL will be malformed and the responding server likely will not process your result. The full syntax for URL encoding is described in RFC 2396.

Once the outgoing data is URL-encoded, it's time for `run` to set up the HTTP connection. It does this using the GCF, setting the request method to `POST`, indicating that the content type being sent is URL-encoded form data as if from a web browser, and specifying the length of the data. (This example *could* have used an HTTP `GET` request and simply tacked the form data on the end of the URL, but it's more useful to show you how to make an HTTP `POST` request, because it's likely you'll be using `POST` for any serious web service interactions anyway.)

With the connection configured, the code writes the arguments to the connection's output stream and proceeds to open and read the result from the input stream. In practice, this is where the code is likely to block; on a mobile wireless device, it may take up to a second or so for this interaction to proceed. In fact, it's quite likely that this transaction will take *longer* to process this interaction, at least the first time the application executes, because many devices, especially MIDP devices, will ask the user to permit the MIDlet if it's OK to perform the request. Only signed third-party applications are permitted to use network resources without permission; others will prompt the user for permission prior to initiating any network request.

Once the code reads the result from the network, it constructs a new `String` with the resulting data and updates the `Location` object with the new data before notifying the application that the `Location` object has changed.

Tip In a crude sense, the `WeatherFetcher` is using the Observer pattern, using the `Location` object it received at initialization as a model, and the application as a receiver of updates. I could have constructed a more elaborate interface and class hierarchy to represent this, although doing so would not have simplified the purpose of this example, which is to show you how to use the `HttpConnection`.

The static `urlEncode` method is trivial, creating a new string based on an input string. The new string has URL-encoded characters for any nonalphabetic characters. Frankly, it's a mystery to me why the CLDC and CDC don't provide this function somewhere as a utility; most network-aware applications are going to need it anyway.

Integrating the `WeatherFetcher` class into the WeatherWidget application does not require much work; starting from the implementation you last saw in Chapter 6, you only need to make a few changes to the MIDlet itself to initiate requests for weather updates when the MIDlet first launches or when you select a new location from the location list. Listing 12-11 shows these changes.

Listing 12-11. *The WeatherWidget MIDlet Modified to Use the WeatherFetcher Class*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class WeatherWidget extends MIDlet implements CommandListener {
    private Form wxForm;
    private StringItem locationItem;
    private StringItem wxItem;
    private Command exitCommand;
    private Command screenCommand;
    private Command settingCommand;
    private Command okCommand;
    private Command backCommand;
    private Command updateCommand;
    private List locationList;
    private TextBox locationTextBox;
    private Alert cannotAddLocationAlert;
    private WeatherFetcher fetcher;
    private Location location;
    private LocationStore locationStore;

    private void initialize() {
        locationStore = new LocationStore();
        String[] locations = locationStore.getLocationStrings();
        try {
            if (locations.length > 0 )
                location = locationStore.getLocation(locations[0]);
        }
        catch(Exception e){}
        fetcher = new WeatherFetcher(location, this);
        getDisplay().setCurrent(get_wxForm());
    }

    public void startApp() {
        initialize();
    }

    public void pauseApp() {
    }
```

```
public void destroyApp(boolean unconditional) {
    if (fetcher!=null) fetcher.cancel();
    fetcher = null;
}

public void update() {
    get_wxItem().setText(get_forecast());
    try
    {
        locationStore.updateLocation(location);
    }
    catch(Exception e){}
    fetcher = null;
}

public void commandAction(Command command, Displayable displayable) {
    // Insert global pre-action code here
    if (displayable == wxForm) {
        if (command == exitCommand) {
            exitMIDlet();
        } else if (command == settingCommand) {
            getDisplay().setCurrent(get_locationList());
        }
} else if (displayable == locationList) {
    if (command == screenCommand) {
            getDisplay().setCurrent(get_locationTextBox());
    } else if (command == List.SELECT_COMMAND) {
            int index = get_locationList().getSelectedIndex();
            set_location(get_locationList().getString(index));
            if (fetcher != null) fetcher.cancel();
            fetcher = new WeatherFetcher(location, this);
            getDisplay().setCurrent(get_wxForm());
    } else if (command == backCommand) {
            getDisplay().setCurrent(get_wxForm());
        }
    } else if (displayable == locationTextBox) {
        if (command == backCommand) {
            getDisplay().setCurrent(get_locationList());
        } else if (command == okCommand) {
            add_location(locationTextBox.getString());
            getDisplay().setCurrent(get_locationList());
        }
```

```
        } else if (displayable == cannotAddLocationAlert) {
            if (command == backCommand) {
                getDisplay().setCurrent(get_locationList());

            }
        }
    }

    public String get_forecast() {
        if (location == null) {
            return "unknown forecast";
        } else {
            return location.getForecast();
        }
    }

    public String get_location() {
        if (location == null) {
            return "unknown";
        } else {
            return location.getLocation();
        }
    }

    public void set_location( String l ) {
        try {
            location = locationStore.getLocation(l);
        }
        catch(Exception e) {}
        get_wxForm().setTitle(l);
    }

    public void add_location( String l ) {
        String locations[];
        int i;
        try {
            locationStore.addLocation( new Location( l, "" ));
        } catch (Exception e) {
            getDisplay().setCurrent(get_cannotAddLocationAlert());
        }
        // Refresh the location list lazily.
        locationList = null;
    }
```

```
public Display getDisplay() {
    return Display.getDisplay(this);
}

public void exitMIDlet() {
    if (fetcher!=null) fetcher.cancel();
    fetcher = null;
    getDisplay().setCurrent(null);
    destroyApp(true);
    notifyDestroyed();
}

public StringItem get_wxItem() {
    if (wxItem == null) {
        wxItem = new StringItem("Forecast", get_forecast());
    }
    return wxItem;
}

public Form get_wxForm() {
    if (wxForm == null) {
        wxForm = new Form(get_location(), new Item[] {
            get_wxItem()
        });
        wxForm.addCommand(get_exitCommand());
        wxForm.addCommand(get_settingCommand());
        wxForm.setCommandListener(this);
    }
    return wxForm;
}

public TextBox get_locationTextBox() {
    if (locationTextBox == null) {
        locationTextBox = new TextBox("Add Location", "", 80, 0);
        locationTextBox.addCommand(get_backCommand());
        locationTextBox.addCommand(get_okCommand());
        locationTextBox.setCommandListener(this);
    }
    return locationTextBox;
}
```

```
    public List get_locationList() {
        if (locationList == null) {
            String[] locations;
            locations = locationStore.getLocationStrings();

            locationList = new List("Where", List.IMPLICIT, locations, null);
            locationList.addCommand(get_screenCommand());
            locationList.addCommand(get_backCommand());
            locationList.setCommandListener(this);
        }
        return locationList;
    }

    public Alert get_cannotAddLocationAlert() {
        if (cannotAddLocationAlert == null)
        {
           cannotAddLocationAlert = new Alert("Cannot Add Location");
           cannotAddLocationAlert.setString("An error occurred adding the➥
location you entered. It has not been added.");
           cannotAddLocationAlert.addCommand(get_backCommand());
        }
        return cannotAddLocationAlert;
    }

    public Command get_settingCommand() {
        if (settingCommand == null) {
            settingCommand = new Command("Settings", Command.OK, 1);
        }
        return settingCommand;
    }

    public Command get_okCommand() {
        if (okCommand == null) {
            okCommand = new Command("OK", Command.OK, 1);
        }
        return okCommand;
    }

    public Command get_exitCommand() {
        if (exitCommand == null) {
            exitCommand = new Command("Exit", Command.EXIT, 1);
        }
        return exitCommand;
    }
```

```
    public Command get_screenCommand() {
        if (screenCommand == null) {
            screenCommand = new Command("Add Location", Command.SCREEN, 1);
        }
        return screenCommand;
    }

    public Command get_backCommand() {
        if (backCommand == null) {
            backCommand = new Command("Back", Command.BACK, 1);
        }
        return backCommand;
    }
}
```

The MIDlet must now declare an instance of `WeatherFetcher`, so that it can initiate and cancel requests for weather updates at launch or when you choose a location. The `initialize` method takes care of the first case, in which the application requests an update for a weather forecast at application launch.

The new `update` method, which the `WeatherFetcher` instance invokes, must do two things: update the user interface with the newly obtained forecast, and update the `LocationStore` with the new weather data. It silently ignores any failures to update the `LocationStore`, under the assumption that the old cached data is likely good enough, and subsequent views of the same location will refetch the weather forecast data anyway.

The MIDlet's main command handler `commandAction` must react when you choose a new location by cancelling any pending operation and creating a new `Fetcher` for the newly selected location, initiating a new network request.

Finally, exiting the MIDlet should dismiss the network operation; this is done by both `destroyApp`, which the application manager will invoke if the device must terminate the application, and `exitMIDlet`, which the user interface itself invokes.

Securing Your HTTP Transaction with HTTPS

While HTTP has many advantages as an application-layer protocol, security's not one of them. Sharing information over HTTP is like talking in a crowded restaurant; the only security you get is through obscurity, and that's not really security at all. To meet the needs of secure applications such as e-commerce, there's the HTTPS protocol, which is simply HTTP implemented over a secure socket connection such as TLS. HTTPS provides both client/server authentication through certificates and encryption of the end-to-end communication using a mutually agreed-upon protocol, defending it against many kinds of attacks, such as man-in-the-middle and eavesdropping attacks. Required by MIDP 2.0, HTTPS is an important addition to the Java ME platform that finds use in many applications.

Fortunately, using HTTPS in your application is as simple as tacking on a couple of `s` characters. When creating the connection, request an `HttpsConnection` from the `Connector` instead, as shown in Listing 12-12.

Listing 12-12. *Requesting an HttpsConnection*

```
HttpsConnection hc = (HttpsConnection)
    Connector.open("https://www.noplace.com/");
```

The resulting `HttpsConnection` implements `HttpConnection`, so you use it the same way. In addition, you can obtain two additional pieces of information about the connection: the port on which the server accepted the connection via its `getPort` method, and details about the negotiated secure connection via its `getSecurityInfo` method.

The `getSecurityInfo` method returns an instance of `SecurityInfo` that describes the kind of connection established between your application and the remote server; from it, you can obtain four pieces of information: how the GCF encrypted the transaction, the bearer protocol, the version of the bearer protocol, and the remote certificate. This information is available via the following `SecurityInfo` methods:

- `getCipherSuite`: Returns a `String` naming the cipher suite that the GCF used to encrypt the transaction
- `getProtocolName`: Returns a `String` indicating the name of the protocol bearing the transaction
- `getProtocolVersion`: Returns a `String` indicating the version of the protocol bearing the transaction
- `getServerCertificate`: Returns the certificate of the remote server

The certificate returned by `getServerCertificate` is an object that implements the `Certificate` interface; you can query it for its properties, such as who issued the certificate and when it will expire.

While adding HTTPS to your application may be easy, crafting a secure application isn't. As leading computer security expert Gene Spafford has remarked, HTTPS is like "using an armored truck to transport rolls of pennies between someone on a park bench and someone doing business from a cardboard box." To craft a truly secure application, you must also take responsibility for the data your application exchanges with remote servers; for example, an e-commerce application should take care to protect a user's credentials to avoid compromise if the user's device is lost or stolen. I show you some of the tools that Java ME provides to meet these challenges in Chapter 15.

Granting Permissions for Network Connections

As I indicated in the section "Putting HTTP to Work," networked applications require privilege. This requirement, imposed by the MIDP, ensures that applications do not generate unauthorized network connections that may result in data use charges or in transmitting your information to unauthorized individuals. Network access through the GCF in the MIDP is guarded by permissions based on the type of connection your application wants to create. Table 12-6 shows the names of these permissions.

Table 12-6. *MIDP Permissions Governing Access to the GCF*

Permission	Interface
`javax.microedition.io.Connector.http`	`HttpConnection`
`javax.microedition.io.Connector.https`	`HttpsConnection`
`javax.microedition.io.Connector.datagram`	`DatagramConnection`
`javax.microedition.io.Connector.datagramreceiver`	`DatagramConnection`
`javax.microedition.io.Connector.socket`	`SocketConnection`
`javax.microedition.io.Connector.serversocket`	`ServerSocketConnection`
`javax.microedition.io.Connector.ssl`	`HttpsConnection`
`javax.microedition.io.Connector.comm`	`CommConnection`
`javax.microedition.io.PushRegistry`	`PushRegistry`

By default, working in the emulator provided by NetBeans, your MIDlet runs as an untrusted MIDlet, and connections are only available if the user grants permission for the connection.

As I describe in Chapter 3, trusted applications must specify the privileges they require in the JAD attributes `MIDlet-Permissions` and `MIDlet-Permissions-Opt`. The easiest way to do this in NetBeans is by specifying this in the application's properties; choose Properties ➤ Application Descriptor ➤ API Permissions and click the button labeled Add. Figure 12-2 shows this screen in NetBeans.

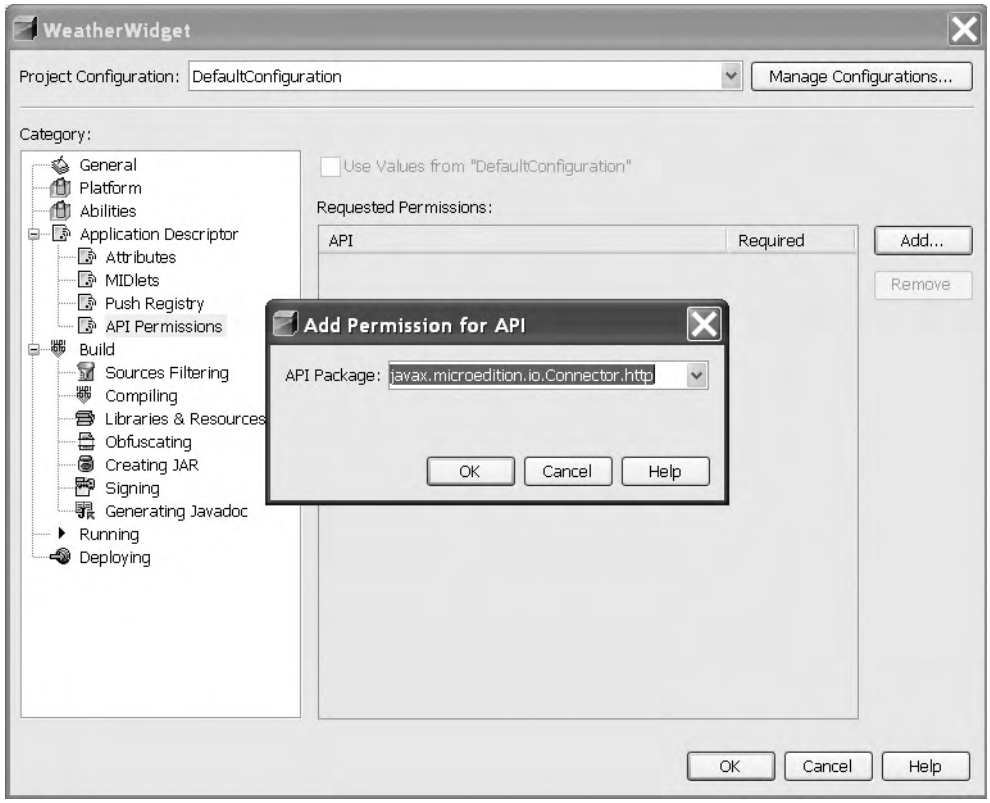


Figure 12-2. *Specifying MIDlet permissions*

Wrapping Up

Unlike other configurations of Java, Java ME provides a unified approach to data communication through the GCF. The GCF provides a factory class for connections, called `Connector`, that lets you create subclasses of the `Connection` interface based on the URL you pass to `Connector.open`. The URL specifies not only the destination of the connection, but also the protocol type and port, letting you create a wide variety of different connections. However, the actual connection types (the application-layer protocols and transport-layer protocols) are limited by the platform on which your application is executing; some protocols such as HTTP are available on nearly the entire range of Java ME platforms including both the CDC and the CLDC, while other protocols such as datagram protocols or low-level streamed socket protocols may not be.

You perform the actual communication using the `Connection` interface subclass that `Connector.open` returns; it typically provides a means for you to obtain either streams or datagrams that you use to exchange bytes with the remote side of the connection. Most subclasses of `Connection` provide additional methods for performing protocol-specific operations, such as setting socket options.

An important example of a `Connection` interface subclass is the hierarchy that begins with `ContentConnection` and contains `HttpConnection`; you use these two classes to perform HTTP exchanges with remote web servers to obtain content and interact with remote web services. Using just the `HttpConnection`, you can post data to a remote server in URL-encoded form and obtain a response that your application can parse, store, and display; you can do the same over HTTPS using the `HttpsConnection`.

When running on the CLDC MIDP platform, your application requires privilege in order to establish a connection. Untrusted applications prompt the user for permission to establish a connection; trusted applications can indicate their need for access to a specific connection type through the JAD attributes `MIDlet-Permissions` and `MIDlet-Permissions-Opt`.

CHAPTER 13

Accessing Web Services

In the last chapter, you learned how to establish connections to remote resources, such as remote web servers. While connecting to remote resources is a crucial component of many Java ME applications, most of today's networked Java ME applications need something more: the ability to interoperate with today's web services, the services provided by web servers designated for machine-to-machine interaction. Web services are an important evolution of the client/server model, and applications with clear user interfaces that present data, such as weather, financial, and travel information, from web services are in high demand as mobile users increasingly rely upon mobile devices for access to products and services.

In this chapter, I show you how to use the optional J2ME Web Services Specification specified by JSR 172 to interface the Java ME platform with the web services powering today's applications, as well as how to accomplish the same when the J2ME Web Services Specification isn't available. I begin with a review of the web service architecture from a client perspective, so you'll understand how working with a web service is different from simply pulling a remote resource over the Internet. In the process, you'll learn the three categories of web service architecture, as well as understand how HTTP and XML play key parts in supporting today's web services. Finally, I discuss in detail the two pieces of the web service puzzle missing from the previous chapter: how to generate and parse XML within a Java ME application.

Looking at a Web Service from the Client Perspective

Most breakthroughs in computing aren't revolutionary, but rather evolutionary. The web service is no different; at its heart, it's an application of the client/server computing model using HTTP as a bearer protocol and a well-known language such as XML, JavaScript Object Notation (JSON), or YAML Ain't Markup Language (YAML) for data representation. A full-fledged web service has three key components, as you see in Figure 13-1:

- *A service broker*: Responsible for helping clients determine the location of one or more *service providers*
- *One or more service providers*: Responsible for providing remote computing facilities such as computation and object exchange to clients
- *One or more service requestors*: The clients that use the broker to find the provider and interact with the provider

---

■ **Note** In the discussion that follows, I use the term *client* to refer to the service requestor.

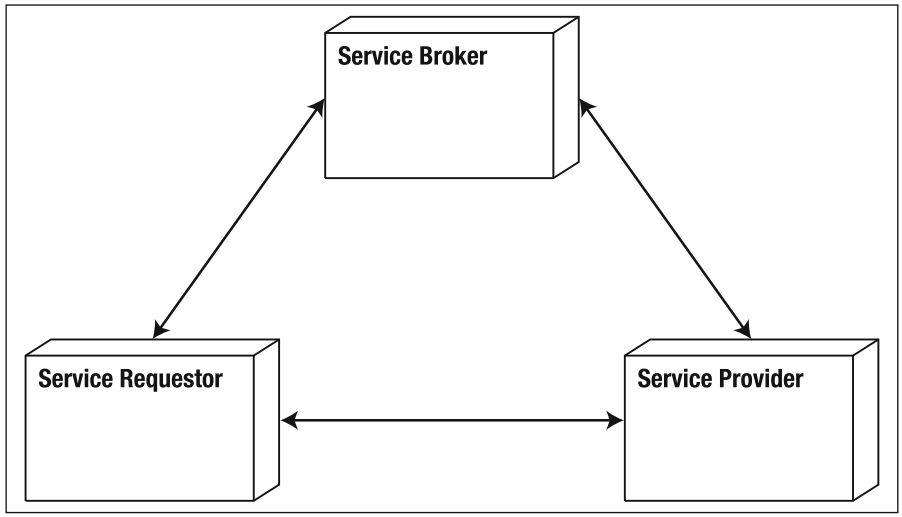


Figure 13-1. *A schematic representation of a web service*

In a classic web service, the requestor first discovers what web services are available by coordinating with the service broker using the Universal Description, Discovery, and Integration (UDDI) registry protocol; once the requestor determines what services are available from a provider, the requestor and provider communicate about the actual services available using the Web Services Description Language (WSDL). Actual requestor-provider communications utilize the Simple Object Access Protocol (SOAP). UDDI, WSDL, and SOAP are all XML applications in that they use XML as the base language for representing data. In practice, HTTP carries the requests and responses that these languages encode, although of course any reliable protocol capable of carrying object data and metadata about objects would suffice.

Considering the Architecture

In practice, the term *web service* has come to apply to just about any machine-to-machine communications application that makes use of web technologies. More than just shorthand to describe a certain class of application, this designation is appropriate as the paradigm has evolved.

In large-scale environments, a web service's division of responsibilities, as shown in Figure 13-1, is important. However, in many situations, the division of responsibilities can be (and has been) simplified to reflect differences in requirements and computing environments.

One of these simplifications is the removal of the broker, or the combination of the broker and service provider. In this model, clients have a predefined notion of the location and nature of the services provided by service providers. This is especially true for many mobile clients, because the complexity and overhead of dynamic discovery and capability negotiation can add significantly to the memory costs of an application.

Another way to categorize web services is by looking at the nature of the interaction between client and server. Broadly speaking, there are three categories of interaction: remote procedure call (RPC) services, service-oriented architecture (SOA) services, and services that use representational state transfer (REST). I like to think of these as taking a functional approach, a messaging approach, and a resource approach, respectively.

RPC-oriented web services are among the most common, and their semantics date back to pre-Web client/server programming in which a client requests a remote server to execute a specific operation. The semantics of encapsulating a request and response typically vary from service to service; small services make great use of simple XML representations for objects and requests, while more complex applications almost always use SOAP to represent objects.

■ **Note** The Java RMI you first saw in Chapter 11 is another example of an RPC interface.

SOA web services are among the largest and most complex, and SOA is widely supported by many industry vendors today. Unlike RPC, in which the primary concept is that of a method call that is typically tightly coupled with the underlying implementation and domain, SOA represents requests as messages. In other words, its focus is on messages, not operations. SOA can provide many advantages when implementing large-scale web services between many different subsystems or organizations. SOA web services use SOAP—an XML representation of objects for data interchange—and WSDL to describe the kinds of services a server or client offers or needs.

Finally, web services based on REST attempt to use the range of methods provided by HTTP to describe the kinds of operations a client may request a service to perform on documented objects. A hallmark of RESTful web services is the presentation of well-known URLs that represent objects, rather than operations, services, or a general

invocation point. Clients request interactions with those objects using HTTP methods such as `GET`, `PUT`, or `DELETE`. RESTful applications may use SOAP, but the strong inclinations toward parsimony in RESTful services lead many architects to use simple, homegrown XML representations of objects.

While large web services use a hierarchy of World Wide Web Consortium (W3C) protocols including UDDI, WSDL, and SOAP, it's entirely possible for you to build a web service using only XML. Many popular Web 2.0 services used in software mashups do this, eschewing the complexity of the full W3C stack for lightweight object representation based solely on SOAP or occasionally just an application of XML alone. Increasingly, web services may provide support for other structured data representations, such as JSON or YAML, enabling software developers to capitalize on the support for these newer data representations in programming languages such as JavaScript or Ruby.

■ **Tip** While there's no reason why you can't build on or use a web service that provides JSON or YAML interfaces with your Java ME application, you should be prepared to do more work to support these data representations, because there's little support on the Java ME platform for them as I write this. I imagine that this situation is likely to change in the coming years as more services use these representations in their client/server communication.

Exchanging Data over the Network

If you're writing a client for any of today's web services, odds are that you will use either HTTP or HTTPS. As I noted in the previous chapter in the section "Securing Your HTTP Transaction with HTTPS," HTTP is a good protocol for data exchange, but it's not terribly secure. Many web services, including those for e-commerce, almost certainly use HTTPS. It's important to remember that not only do you need to protect the security of your users' data with HTTPS, but you also need to take responsibility for protecting their data elsewhere in your application, such as when it's stored in a record store or file.

Another consideration when writing web service clients is the HTTP methods you may need to use; this is especially true when using REST, because the REST paradigm relies on the HTTP method to provide some of the semantics about a web service operation. For example, in writing a web service client to a messaging application that's built around REST, you may find that the web service definition for deleting a message requires that you send an HTTP request with the `DELETE` method. Fortunately, this is as easy as specifying the method using the `HttpConnection`'s (or `HttpsConnection`'s) `setRequestMethod` passing the method you want to use, as shown in Listing 13-1.

Listing 13-1. *Specifying the HTTP Method*

```
try {
    hc = (HttpConnection)Connector.open(url);
    hc.setRequestMethod("DELETE");
    …
} catch (Exception e) {…}
```

You can actually pass *any* string to `setRequestMethod`, but you should of course stick to the methods defined by HTTP, which I summarize for you in Table 12-4 in Chapter 12.

Another consideration is the need some web services have for *message digests*, which are secure hashes of the web request or object body from the client to the server. I discuss how to compute message digests using various options available to Java ME developers in Chapter 15.

One final thing you may encounter when developing a client for a web service at the HTTP layer is a need for a cookie to save session state. This is especially true for SOA-derived web services; often these services require the client application to log in and present some state data provided during the login sequence during each transaction. This is usually done by means of a *cookie*—a bit of data provided by the server in an HTTP header that encapsulates a session with the server. Web clients use cookies all the time to store state data between the client and server; your application may be called upon to do the same thing. The full details of how HTTP can use cookies to carry session state are described in RFC 2965 and RFC 2109. However, you should know the following facts:

- Web servers deliver cookies to your application using the `Set-Cookie` header.
- Cookies come in several parts: the first part is the session ID of the cookie, which is the data you need to replay to a server in subsequent requests, and the remainder is metadata about the cookie, such as when it expires and to which domain it applies. Semicolons separate these parts.
- You deliver a cookie to the web server by using the `Cookie` header.

To look for cookies from a server, you simply use the `HttpConnection`'s `getHeaderField` method, as shown in Listing 13-2.

Listing 13-2. *Looking for Cookies from a Server*

```
String cookie = hc.getHeaderField("Set-Cookie");
if (cookie != null) {
  int semicolon = cookie.indexOf(';');
  /* session is at cookie.substring(0, semicolon); */
}
```

Once you get a cookie, you should break it into its session ID and domain parts, which you can do by cracking the cookie string at its semicolon; this splits the two parts. A good place to store the resulting session is in a member field of the Java class implementing your web service client. You'll then check the domain against subsequent URLs you request, sending the cookie only if the domains match using the `HttpConnection`'s `setRequestProperty`, as shown in Listing 13-3.

Listing 13-3. *Sending a Cookie with an HTTP Request*

```
HttpConnection hc = (HttpConnection)Connector.open(url);
hc.setRequestProperty("cookie", mSession);
```

You may want to store cookies elsewhere, such as the record store, but there's a catch: they expire, and when they expire depends on how the server is configured. If you want to persist cookies, you should definitely parse the cookie for the expiration date and handle expirations appropriately by removing them from your application's store when they expire.

Using XML for Data Representation

XML provides a standard syntax for creating custom markup languages that let you describe the data within a document. Derived from the Standard Generalized Markup Language (SGML), it's one of the most widely used markup languages today. In this section, I give you just a thumbnail sketch of XML, the nitty-gritty you need to understand to use simple XML in your own applications. For a thorough introduction to XML, I encourage you to visit the W3C documentation at `http://www.w3c.org/XML`.

XML syntax provides you with a general syntax for representing document trees that originate with a single *root* element that must contain zero or more *child* elements, which themselves can have children. Elements consist of *tags* that have attributes and may contain other tags or data, like the example of a weather report shown in Listing 13-4.

Listing 13-4. *A Sample XML Document*

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A simple weather example -->
<weather city="Berkeley" state="CA" country="USA">
    <temperatures units="F">
        <temperature type="high">76</temperature>
        <temperature type="low">56</temperature>
        <temperature type="current">72</temperature>
    </temperatures>
    <wind>
        <direction units="compass">ENE</direction>
        <speed units="MPH">5</speed></wind>
    <precipitation probability="0" type="rain" units="in"/>
    <text>
        <when time="today">
Mostly cloudy in the morning then becoming mostly sunny.
Patchy fog in the morning. Highs in the 70s to lower 80s.
Afternoon seabreeze 10 to 20 mph.
        </when>
        <when time="tonight">
Mostly clear except areas of low clouds and fog developing
overnight. Lows in the mid to upper 50s. Evening seabreeze
10 to 20 mph.
        </when>
    </text>
</weather>
```

Figure 13-2 shows a tree representing the structure of this data.

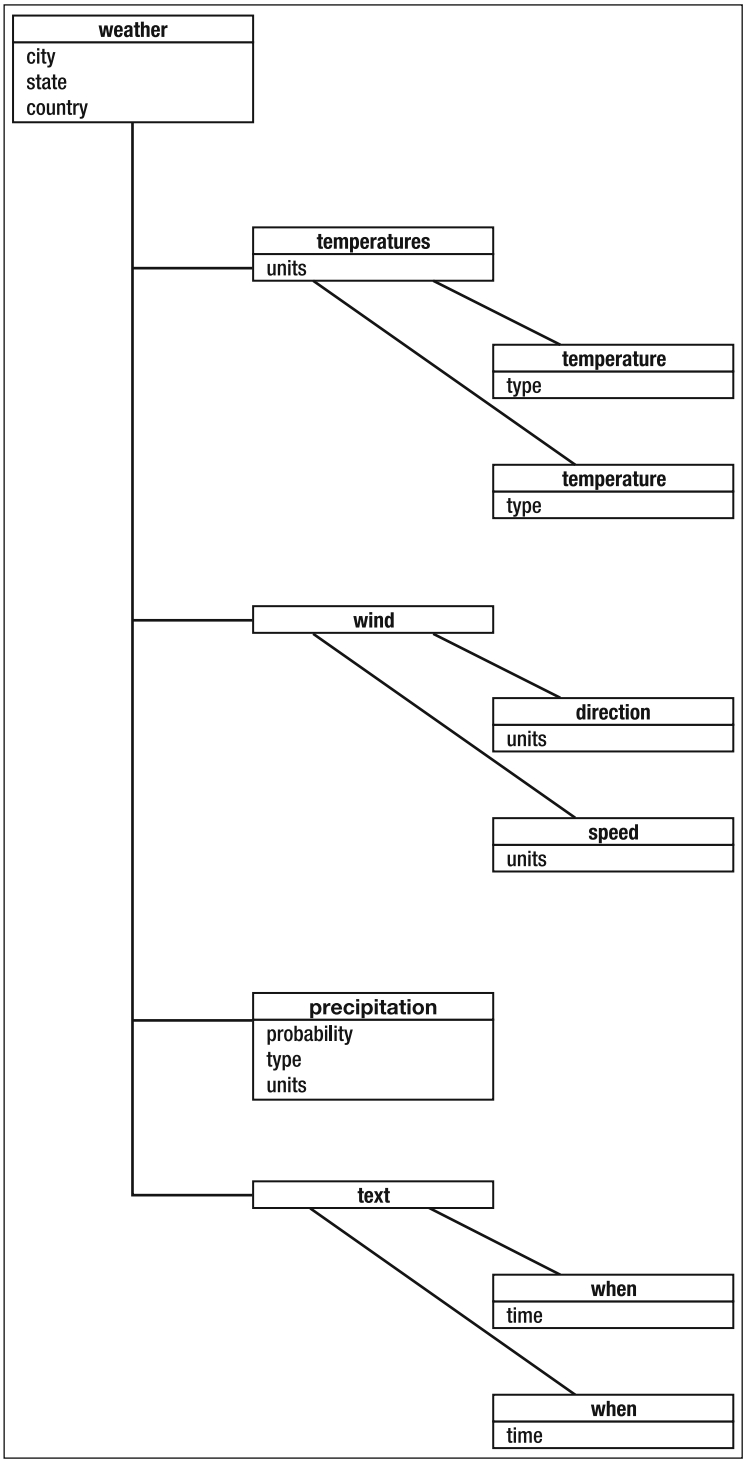


Figure 13-2. *A tree representing the structure of the XML document in Listing 13-4*

This example shows several key features of XML:

- XML documents begin with a *preamble*, delimited by <? and ?>. The preamble indicates the version of XML used by the file and the character set encoding for the file.
- XML documents may contain comments that begin with <!-- and end with -->. (Comments cannot be nested, however.)
- Elements consist of plain text contained by < and >. (Actually, there are limits to what can go between these symbols; for this discussion, it's safe to assume that the name of an element must consist of only alphanumeric characters and the underscore character.)
- A starting tag indicates the beginning of an element's data; an ending tag is denoted by the same tag name contained by </ and >.
- White space is not significant between elements.
- Elements may have *attributes*, which are name-value pairs contained within the beginning tag.
- Elements are case-sensitive.
- Empty elements contain no elements and are denoted using < and />.
- Elements may contain other elements or textual data.

■ **Note** I've made up the schema for weather data this chapter uses in order to show a sampling of key features of XML. On the Internet, several schemas for weather data exist, such as the one defined by the National Weather Service of the U.S. government described at `http://www.weather.gov/data/current_obs/` or the Yahoo! weather format described at `http://developer.yahoo.com/weather/`.

XML documents can represent text in one of two ways: as flat character data that may include five *entities* that represent the special characters <, >, &, ', and ", or as a special unparsed character directive that contains raw character data that the XML parser accepts without attempting to parse. Table 13-1 shows the five character entity definitions.

Table 13-1. *XML Character Entities*

Entity	Character	Represents
&	&	Ampersand
<	<	Less than
>	>	Greater than
'	'	Apostrophe
"	"	Quotation mark

Listing 13-5 shows a simple unparsed character directive example.

Listing 13-5. *An Example of Unparsed Character Data Using XML's CDATA Directive*

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
<![CDATA[
Every character within the CDATA construct is treated as a
regular character; there are no restrictions, so I can write
things like 3 < 4 without confusing the XML parser.
]]>
</root>
```

An XML document may define one or more *namespaces* that define uniquely named elements and attributes within the document. This permits an XML document to use a single element or attribute for multiple purposes, such as an `<id>` element that pertains to both a customer and a product ID. When using namespaces, you use the XML-restricted `xmlns` tag to define a unique URL that identifies the namespace; namespace names precede the tag name in tags. You separate the namespace name from the tag name using a colon, as Listing 13-6 shows.

Listing 13-6. *An Example of XML Namespaces*

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A simple weather example -->
<wx:weather city="Berkeley" state="CA" country="USA"
 xmlns:wx="www.apress.com/rischpater/weather">
    <wx:temperatures units="F">
        <wx:temperature type="high">76</wx:temperature>
        <wx:temperature type="low">56</wx:temperature>
        <wx:temperature type="current">56</wx:temperature>
    </wx:temperatures>
```

```
    <wx:wind units="MPH">
        <wx:direction units="compass">ENE</wx:direction>
        <wx:speed>5</wx:speed></wx:wind>
    <wx:precipitation/>
    <wx:text>
        <wx:when time="today">
Mostly cloudy in the morning then becoming mostly sunny.
Patchy fog in the morning. Highs in the 70s to lower 80s.
Afternoon seabreeze 10 to 20 mph.
        </wx:when>
        <wx:when time="tonight">
Mostly clear except areas of low clouds and fog developing
overnight. Lows in the mid to upper 50s. Evening seabreeze
10 to 20 mph.
        </wx:when>
    </wx:text>
</wx:weather>
```

XML documents provide levels of validity; *well-formed* documents are those that meet the syntactic requirements of XML itself, while *valid* documents meet additional semantic constraints imposed by either users or another XML representation of a document's semantics, such as an XML schema or XML Document Type Definition (DTD). In practice, when using XML for the lightweight clients running on Java ME–enabled devices, you will create and manipulate well-formed XML content while leaving true validation to the server and how you parse the XML itself.

■ **Note** In recent years, XML has been largely adopted as a panacea for data representation challenges, with YAML and JSON as its close compatriots for web-based programming. There's still room, however, for binary-based protocols such as Wireless Application Protocol (WAP) Binary XML (WBXML, the binary version of XML the W3C details at `http://www.w3.org/TR/wbxml/`) and tag-based representations that provide more compact representation of data, especially for applications running over slower networks or applications that charge a premium for data delivery.

Exploring XML Support for Web Services in Java ME

Using a web service from your Java ME application typically involves the following steps:

1. Your application presents a user interface.

2. As a result of some action that the user or application takes, the application must make a web service request to a remote web service.

3. The application generates an XML document representing the web service request (often referred to as *marshalling* data for the request).

4. Using HTTP, the application transmits the XML document to the remote web service via a `POST` request.

5. Using HTTP, the application receives a response XML document from the remote web service.

6. The application parses the resulting XML document to obtain values of interest for the application or user (often referred to as *unmarshalling* the response).

7. The application uses the data it obtained from parsing the resulting XML document.

Thus, web services pose three common programming tasks for Java ME developers that are amenable to generalization across multiple applications through the addition of new APIs:

- Interaction with a remote server using HTTP (as you saw in Chapter 12, the GCF and the `HttpConnection` and `HttpsConnection` handle this)

- Generation of an XML document—typically represented as an instance of `String`—from Java objects

- Parsing of an XML document to obtain values to place in fields of one or more Java objects

The GCF ably addresses the first problem; in addition, there's support for encapsulating the GCF functionality in the J2ME Web Services Specification, which I say more about in the "Introducing the J2ME Web Services Specification" section later in this chapter. XML generation in Java ME—the second problem—typically takes one of two forms: simple concatenation using template components and an instance of the `StringBuffer` class, or use of the SOA-supporting J2ME Web Services Specification. I discuss each of these in subsequent sections of this chapter. You can solve the third problem—parsing XML—in one of two ways, by either relying on the optional Web Services Specification or including a lightweight parser like kXML in your application.

XML parsers come in three varieties: *DOM* parsers, *push* parsers, and *pull* parsers. DOM parsers parse an entire document and return an instance of a document tree that lets you query for elements of the tree using its methods; they are typically expensive both to implement and represent the document tree, and they're generally not used in the Java ME environment. Push parsers call a client interface's methods with XML events, such as the definition of a tag; in essence, they "push" data from the XML

to a specific parser class written to parse XML with specific tags. Pull parsers work the opposite way; the code scanning the XML invokes the XML parser's methods to obtain individual tags and other information. As you see in subsequent sections of this chapter, both the push and pull models are available for Java ME, and which you use is primarily a decision based on the availability of the parser you want to use. In practice, I think it's fair to say that a pull parser may use slightly less memory than a push parser; however, given one, you can easily code the other (as I show later in this chapter), so they're essentially equivalent in all but name and how you structure the code that interacts with them.

Generating XML in Java ME Applications

Generating XML in a Java ME application is a simple, if tedious, task, using the `StringBuffer` class. You simply build the XML document, tag by tag, using a combination of compile-time strings for the tags and values of your application's objects for the tag and attribute values. Listing 13-7 shows a revised version of the WeatherWidget's `Location` class that supports the additional weather information that an XML document such as the one in Listing 13-4 might bear, including a `toXml` method that creates an XML representation of the `Location` class.

Listing 13-7. *A Version of the Location Class That Can Generate an XML Representation of Its Value*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.rms.*;
import java.io.*;

public class Location {
    private final static int FIELD_VERSION = 1;
    private final static int FIELD_CITY = 2;
    private final static int FIELD_FORECAST = 3;
    private final static int FIELD_TEMP = 4;
    private final static int FIELD_WINDSPEED = 5;
    private final static int FIELD_WINDDIR = 6;
    private final static int FIELD_PRECIP = 7;
    private final static int FIELD_STATE = 8;
    private final static int FIELD_COUNTRY = 9;
    private final static int FIELD_PRECIP_PROB = 10;
    private final static int FIELD_PRECIP_TYPE = 11;
```

```
    private final static String XML_PREAMBLE =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n";
    private final static String XML_START_OPEN = "<";
    private final static String XML_START_EMPTYOPEN = "<";
    private final static String XML_START_CLOSE="</";
    private final static String XML_END_OPEN = ">";
    private final static String XML_END_EMPTYOPEN = "/>";
    private final static String XML_END_CLOSE = ">";
    private final static String XML_ATTR_IS = "=";
    private final static String XML_ATTR_QUOTE = "\"";
    private final static String XML_NEWLINE="\n";

    public final static String XML_TAG_WEATHER="weather";
    public final static String XML_ATTR_CITY="city";
    public final static String XML_ATTR_STATE="state";
    public final static String XML_ATTR_COUNTRY="country";
    public final static String XML_TAG_TEMPS="temperatures";
    public final static String XML_ATTR_UNITS="units";
    public final static String XML_TAG_TEMP="temperature";
    public final static String XML_ATTR_TYPE="type";
    public final static String XML_TAG_WIND="wind";
    public final static String XML_TAG_DIRECTION="direction";
    public final static String XML_TAG_SPEED="speed";
    public final static String XML_TAG_PRECIP="precipitation";
    public final static String XML_ATTR_PROB="probability";
    public final static String XML_TAG_TEXT="text";
    public final static String XML_TAG_WHEN="when";
    public final static String XML_ATTR_TIME="time";

    private final static String XML_ATTR_WTIME=" time=\"today\"";
    private final static String XML_ATTR_TUNITS=" units=\"F\"";
    private final static String XML_ATTR_WUNITS=" units=\"MPH\"";
    private final static String XML_ATTR_PUNITS=" units=\"in\"";

    public final static int NO_ID = -1;

    private final static int version = 1;
    private String city, state, country;
    private String forecast;
    private String temp;
    private String windSpeed;
    private String windDirection;
    private String precipitation, precipitationProb, precipitationType;
    private int recordid;
```

```
/** Creates a new instance of Location */
public Location() {
    recordid = NO_ID;
}

public Location(String l) {
    setLocation(l);
    recordid = NO_ID;
}

public Location(byte[] b) {
    fromBytes(b);
    recordid = NO_ID;
}

public Location(byte[] b, int id) {
    fromBytes(b);
    recordid = id;
}

public String getLocation() {
    StringBuffer result = new StringBuffer();
    if (city!=null) {
        result.append(city);
    }
    result.append(",");
    if (state!=null) {
        result.append(state);
    }
    result.append(",");
    if (country!=null) {
        result.append(country);
    }
   return result.toString();
}

public String getCity() {
    if(city != null)
        return city;
    else
        return "";
}
```

```
public String getState() {
    if(state != null)
        return state;
    else
        return "";
}

public String getCountry() {
    if(country != null)
        return country;
    else
        return "";
}

public void setLocation(String l) {
    int cityStart, stateStart, countryStart;
    cityStart = 0;
    stateStart = l.indexOf(",");
    if (stateStart==-1) {
        city = l;
        state="";
        country="";
        return;
    }
    countryStart = l.indexOf(",",stateStart+1);
    city = l.substring(cityStart,stateStart);
    if (countryStart==-1)
    {
        state = l.substring(stateStart-1);
        country = "USA";
        return;
    }
    state = l.substring(stateStart+1,countryStart);
    country = l.substring(countryStart+1);
}

public void setLocation(String c, String s, String co) {
    city = c;
    state = s;
    country = co;
}
```

```
public String getWind() {
    if (windSpeed != null && windDirection != null)
        return windSpeed + " mph from the " + windDirection;
    else return "";
}

public void setWind(String s, String d) {
    windSpeed = s;
    windDirection = d;
}

public String getTemperature() {
    if (temp!=null)
        return temp;
    else
        return "";
}

public void setTemperature(String t) {
    temp = t;
}

public String getPrecipitation() {
    StringBuffer result = new StringBuffer();
    if (precipitation != null) {
        result.append(precipitation);
    } else {
        result.append("unknown");
    }
    result.append("in of ");
    if (precipitationType != null) {
        result.append(precipitationType);
    } else {
        result.append("unknown");
    }
    result.append("(");
    if (precipitationProb != null) {
        result.append(precipitationProb);
    } else {
        result.append("unknown");
    }
    result.append("%)");
    return result.toString();
}
```

```
    public void setPrecipitation(String p, String pp, String pt) {
        precipitation = p;
        precipitationProb = pp;
        precipitationType = pt;
    }

    public String getForecast() {
        if (forecast != null) {
            return forecast;
        } else {
            return "";
        }
    }

    public void setForecast(String f) {
        forecast = f;
    }

    public int getId() {
        return recordid;
    }

    public void setId(int id) {
        recordid = id;
    }

    public byte[] toBytes() {
        byte[] b;
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);


        // Record format is field-tag, then field for each
        try {
            dos.writeInt(FIELD_VERSION);
            dos.writeInt(version);
            if (city != null) {
                dos.writeInt(FIELD_CITY);
                dos.writeUTF(getCity());
            }
            if (state != null) {
                dos.writeInt(FIELD_STATE);
                dos.writeUTF(getState());
            }
```

```
            if (country != null) {
                dos.writeInt(FIELD_COUNTRY);
                dos.writeUTF(getCountry());
            }
            if (forecast != null)
            {
                dos.writeInt(FIELD_FORECAST);
                dos.writeUTF(getForecast());
            }
            if (temp != null) {
                dos.writeInt(FIELD_TEMP);
                dos.writeUTF(getTemperature());
            }
            if (precipitation != null &&
                precipitationProb != null &&
                precipitationType != null) {
                dos.writeInt(FIELD_PRECIP);
                dos.writeUTF(precipitation);
                dos.writeInt(FIELD_PRECIP_PROB);
                dos.writeUTF(precipitationProb);
                dos.writeInt(FIELD_PRECIP_TYPE);
                dos.writeUTF(precipitationType);
            }
            if (windDirection != null && windSpeed != null) {
                dos.writeInt(FIELD_WINDDIR);
                dos.writeUTF(windDirection);
                dos.writeInt(FIELD_WINDSPEED);
                dos.writeUTF(windSpeed);
            }
        }
        catch(Exception e) {
            return null;
        }

        // Get the bytes for this item.
        b = baos.toByteArray();
        dos = null;
        baos = null;

        return b;
    }
```

```
    public void fromBytes(byte[] b) {
        ByteArrayInputStream bais = new ByteArrayInputStream(b);
        DataInputStream dis = new DataInputStream(bais);
        String dir = null, speed = null;
        String c = null, s = null, co = null;
        String p = null, pp = null, pt = null;
        // Read each tag, then each field
        try
        {
            while(true) {
                int tag = dis.readInt();
                switch(tag) {
                    case FIELD_VERSION:
                        // Don't check version; there's only one
                        dis.readInt();
                        break;
                    case FIELD_CITY:
                        c = dis.readUTF();
                        break;
                    case FIELD_STATE:
                        s = dis.readUTF();
                        break;
                    case FIELD_COUNTRY:
                        co = dis.readUTF();
                        break;
                    case FIELD_FORECAST:
                        setForecast(dis.readUTF());
                        break;
                    case FIELD_TEMP:
                        setTemperature(dis.readUTF());
                        break;
                    case FIELD_PRECIP:
                        p = dis.readUTF();
                        break;
                    case FIELD_PRECIP_PROB:
                        pp = dis.readUTF();
                        break;
                    case FIELD_PRECIP_TYPE:
                        pt = dis.readUTF();
                        break;
                    case FIELD_WINDDIR:
                        dir = dis.readUTF();
                        break;
```

```
                case FIELD_WINDSPEED:
                    speed = dis.readUTF();
                    break;
            }
        }
    }
    catch (Exception e) {}
    finally {
        setLocation(c, s, co);
        setPrecipitation(p, pp, pt);
        setWind(speed, dir);
        try {
            dis.close();
            bais.close();
        }
        catch (Exception e) {}
    }
    dis = null;
    bais = null;
}

public String toXml() {
    StringBuffer xmlBuffer = new StringBuffer();
    xmlBuffer.append(XML_PREAMBLE);
    xmlBuffer.append(XML_START_OPEN);
    xmlBuffer.append(XML_TAG_WEATHER);
    if (city != null) {
        xmlBuffer.append(" ");
        xmlBuffer.append(XML_ATTR_CITY);
        xmlBuffer.append(XML_ATTR_IS);
        xmlBuffer.append(XML_ATTR_QUOTE);
        xmlBuffer.append(city);
        xmlBuffer.append(XML_ATTR_QUOTE);
    }
    if (state != null) {
        xmlBuffer.append(" ");
        xmlBuffer.append(XML_ATTR_STATE);
        xmlBuffer.append(XML_ATTR_IS);
        xmlBuffer.append(XML_ATTR_QUOTE);
        xmlBuffer.append(state);
        xmlBuffer.append(XML_ATTR_QUOTE);
    }
```

```
        if (country != null) {
            xmlBuffer.append(" ");
            xmlBuffer.append(XML_ATTR_COUNTRY);
            xmlBuffer.append(XML_ATTR_IS);
            xmlBuffer.append(XML_ATTR_QUOTE);
            xmlBuffer.append(country);
            xmlBuffer.append(XML_ATTR_QUOTE);
        }
        if (forecast != null ||
            temp != null ||
            (precipitation != null &&
             precipitationProb != null &&
             precipitationType != null) ||
             (windSpeed != null && windDirection != null)) {
            xmlBuffer.append(XML_END_OPEN);
            xmlBuffer.append(XML_NEWLINE);
            if (temp != null) {
                xmlBuffer.append(XML_START_OPEN);
                xmlBuffer.append(XML_TAG_TEMPS);
                xmlBuffer.append(XML_ATTR_TUNITS);
                xmlBuffer.append(XML_END_OPEN);
                xmlBuffer.append(XML_NEWLINE);
                xmlBuffer.append(XML_START_OPEN);
                xmlBuffer.append(XML_TAG_TEMP);
                xmlBuffer.append(" ");
                xmlBuffer.append(XML_ATTR_TYPE);
                xmlBuffer.append(XML_ATTR_IS);
                xmlBuffer.append("\"current\"");
                xmlBuffer.append(XML_END_OPEN);
                xmlBuffer.append(temp);
                xmlBuffer.append(XML_START_CLOSE);
                xmlBuffer.append(XML_TAG_TEMP);
                xmlBuffer.append(XML_END_CLOSE);
                xmlBuffer.append(XML_NEWLINE);
                xmlBuffer.append(XML_START_CLOSE);
                xmlBuffer.append(XML_TAG_TEMPS);
                xmlBuffer.append(XML_END_CLOSE);
                xmlBuffer.append(XML_NEWLINE);
            }
            if (windSpeed != null && windDirection != null) {
                xmlBuffer.append(XML_START_OPEN);
                xmlBuffer.append(XML_TAG_WIND);
```

```
            xmlBuffer.append(XML_ATTR_WUNITS);
            xmlBuffer.append(XML_END_OPEN);
            xmlBuffer.append(XML_NEWLINE);
            xmlBuffer.append(XML_START_OPEN);
            xmlBuffer.append(XML_TAG_SPEED);
            xmlBuffer.append(XML_END_OPEN);
            xmlBuffer.append(windSpeed);
            xmlBuffer.append(XML_START_CLOSE);
            xmlBuffer.append(XML_TAG_SPEED);
            xmlBuffer.append(XML_END_CLOSE);
            xmlBuffer.append(XML_START_OPEN);
            xmlBuffer.append(XML_TAG_DIRECTION);
            xmlBuffer.append(XML_END_OPEN);
            xmlBuffer.append(windDirection);
            xmlBuffer.append(XML_START_CLOSE);
            xmlBuffer.append(XML_TAG_DIRECTION);
            xmlBuffer.append(XML_END_CLOSE);
            xmlBuffer.append(XML_NEWLINE);
            xmlBuffer.append(XML_START_CLOSE);
            xmlBuffer.append(XML_TAG_WIND);
            xmlBuffer.append(XML_END_CLOSE);
            xmlBuffer.append(XML_NEWLINE);
        }
        if (precipitation != null &&
            precipitationProb != null &&
            precipitationType != null) {
            xmlBuffer.append(XML_START_OPEN);
            xmlBuffer.append(XML_TAG_PRECIP);
            xmlBuffer.append(XML_ATTR_PUNITS);
            xmlBuffer.append(" ");
            xmlBuffer.append(XML_ATTR_PROB);
            xmlBuffer.append(XML_ATTR_IS);
            xmlBuffer.append(XML_ATTR_QUOTE);
            xmlBuffer.append(precipitationProb);
            xmlBuffer.append(XML_ATTR_QUOTE);
            xmlBuffer.append(" ");
            xmlBuffer.append(XML_ATTR_TYPE);
            xmlBuffer.append(XML_ATTR_IS);
            xmlBuffer.append(XML_ATTR_QUOTE);
            xmlBuffer.append(precipitationType);
            xmlBuffer.append(XML_ATTR_QUOTE);
            xmlBuffer.append(XML_END_OPEN);
```

```
                xmlBuffer.append(precipitation);
                xmlBuffer.append(XML_START_CLOSE);
                xmlBuffer.append(XML_TAG_PRECIP);
                xmlBuffer.append(XML_END_CLOSE);
                xmlBuffer.append(XML_NEWLINE);
            }
            if (forecast != null) {
                xmlBuffer.append(XML_START_OPEN);
                xmlBuffer.append(XML_TAG_TEXT);
                xmlBuffer.append(XML_END_OPEN);
                xmlBuffer.append(XML_NEWLINE);
                xmlBuffer.append(XML_START_OPEN);
                xmlBuffer.append(XML_TAG_WHEN);
                xmlBuffer.append(XML_ATTR_WTIME);
                xmlBuffer.append(XML_END_OPEN);
                xmlBuffer.append(forecast);
                xmlBuffer.append(XML_START_CLOSE);
                xmlBuffer.append(XML_TAG_WHEN);
                xmlBuffer.append(XML_END_CLOSE);
                xmlBuffer.append(XML_NEWLINE);
                xmlBuffer.append(XML_START_CLOSE);
                xmlBuffer.append(XML_TAG_TEXT);
                xmlBuffer.append(XML_END_CLOSE);
                xmlBuffer.append(XML_NEWLINE);
            }
        xmlBuffer.append(XML_START_CLOSE);
        xmlBuffer.append(XML_TAG_WEATHER);
        xmlBuffer.append(XML_END_CLOSE);
        xmlBuffer.append(XML_NEWLINE);
        } else {
            xmlBuffer.append(XML_END_EMPTYOPEN);
        }

        return xmlBuffer.toString();
    }

    public void fromXml(String xml) {
        /* shown in Listing 13-9 and 13-13 */
    }
}
```

The code begins with static definitions of the various fields the object now stores, along with static `String` declarations for the various bits of XML the code uses when building up its XML representation. Much of the class remains unchanged from Chapter 6, except the addition of new fields and accessor/mutator methods for these fields; the `toBytes` and `fromBytes` methods have changed to serialize the new fields using the same tag-value system I introduced in the discussion about record stores in Chapter 6. The `toXml` method constructs the XML in a `StringBuffer` instance by appending the preamble and bits of static XML along with the values of each member variable. This technique is crude, but it works well; depending on the kind of object you want to represent as XML, you can often streamline the code a bit, such as when you have vectors of similar objects.

If you've spent a fair amount of time using Java, you might look at the `toXml` method and be tempted to do something similar with Java reflection, creating a generic XML encoder object that takes an instance of an object, interrogates the object for member fields, and writes XML based on the type and contents of each field. This is an excellent approach for Java SE, because Java SE provides reflection, but unfortunately the approach falls short on Java ME devices, because as I discuss in Chapter 2, the CLDC lacks support for reflection. If your code is destined to run only on the CDC, a reflection-oriented approach may well make sense.

Introducing the J2ME Web Services Specification

The J2ME Web Services Specification (as its name indicates, its development predates the Java ME platform, and it's an option for both CLDC- and CDC-based devices) that JSR 172 describes provides two additions to the Java ME platform: access to remote SOAP-based web services, and a push XML parser. These additions—collectively called the optional Web Services Specification—are both optional; a platform implementing JSR 172 can offer SOAP-based web services support, a push XML parser, or both. Many advanced mobile devices today include both portions of JSR 172, although it's not available everywhere; as with other optional specifications, it's important to confirm the availability of the specified API on your device targets early in product development.

The implementation of SOAP-based web services support is a subset of Java API for XML-based RPC (JAX-RPC), the Java interface to web-based RPC services. This subset provides the client implementation of JAX-RPC, including marshalling and unmarshalling (between Java and SOAP) of `boolean`, `byte`, `short`, `int`, `long`, `String`, arrays, and complex types and static stub-based invocation of RPC calls by the client. Due to platform limitations, it's not a full implementation of JAX-RPC; devices implementing the specification cannot offer service end points or use extensible type mapping.

Applications using the JAX-RPC support must follow these steps:

1. At compile time, define and generate a stub from the WSDL description of the service.

2. At runtime, instantiate an instance of the stub.

3. At runtime, invoke methods on the stub corresponding to the service end point's operational implementation.

4. Package the generated stub with the Java ME client application.

The work here is similar to what you need to do if you want to use the optional support for Java RMI that I describe in Chapter 11, because the JAX-RPC interface uses the same overall architecture, relying on a local stub object to provide an interface to the remote service provider.

While the JAX-RPC interface provides an important option available to you when interfacing with some existing legacy services, a growing number of web services are moving toward or fully implement a RESTful approach, in which the relatively heavy semantics of SOAP is replaced by a more descriptive use of XML. In a way, this is turning the use of XML back to its origins, in which it's used to provide a human- and machine-readable description of the contents of a data object. In that regard, it's the second optional component of JSR 172 that becomes important to you: the XML parser.

The XML parser that JSR 172 defines is a strict subset of the XML parser defined in JSR 63. It specifies that

- The implementation must not provide any support for the Simple API for XML (SAX) parsing.
- The implementation must not provide any support for DOM, because DOM is too heavy in terms of implementation size and runtime memory footprint for use on Java ME devices.
- The implementation must not support XML Transformations (XSLT).
- The implementation is not required to support XML validation, as validation is expensive in terms of processing power and memory.
- The implementation must support the predefined XML entities (e.g., `&`) but is not required to support optional XML entities.
- The implementation must support XML namespaces.
- The implementation must support both UTF-8 and UTF-16 character encodings.

The optional parser implementation includes three packages: `javax.xml.parsers`, `org.xml.sax`, and `org.xml.sax.helpers`. The first package contains the platform's parser implementation itself, while the other packages provide utilities for the parser. The parser implementation must implement the SAX2 parsing interface, as reflected by the namespace for the packages.

SAX provides a straightforward interface to parsing XML, in which the parser consumes XML from a stream and generates events during parsing that represent specific

XML elements, comments, and other items. In using a SAX parser, you provide an event handler that accepts the events the SAX parser sends and builds the resulting object from those events. Your event handler implements the interface defined by the `org.xml.sax.helpers.DefaultHandler` class, overriding its methods to handle individual SAX events. Some of the events you can choose to handle include the following:

- `startDocument`: Signals the beginning of the XML document.
- `endDocument`: Signals the end of the XML document.
- `startElement`: Signals the start of an element and is given the element's name, namespace, and attributes. Attributes are passed using an instance of an `Attributes` class that provides a collection of all of the attributes of an element, making it easy for you to find the value of a specific attribute.
- `endElement`: Signals the end of an element.
- `characters`: Takes character data from within an element.

Returning to the WeatherWidget example, Listing 13-8 shows the `LocationParserHandler` class—a subclass of `DefaultHandler` that parses an XML document representing a `Location` object with its associated forecast.

Listing 13-8. *The LocationParserHandler SAX Handler*

```
package com.apress.rischpater.weatherwidget;

import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class LocationParserHandler extends DefaultHandler {
    private Location location;
    private boolean setForecast;
    private boolean hasForecast;
    private boolean setTemp;
    private String precipitation, precipType, precipProb;
    private String windSpeed, windDirection;
    private StringBuffer buffer;

    public LocationParserHandler(Location l) {
        location = l;
    }
```

```
public Location getLocation() {
    return location;
}

public void startDocument() {
    if (location == null)
        location = new Location();
    setForecast = false;
    hasForecast = false;
    setTemp = false;
}

public void startElement(String uri, String localName,
    String qName, Attributes attributes)
    throws SAXException {
    buffer = new StringBuffer();
    if (qName.equals(Location.XML_TAG_WEATHER) == 0) {
        String c = attributes.getValue(Location.XML_ATTR_CITY);
        String s = attributes.getValue(Location.XML_ATTR_STATE);
        String co = attributes.getValue(Location.XML_ATTR_COUNTRY);
        location.setLocation(c,s,co);
        hasForecast = false;
    }
    if (qName.equals(Location.XML_TAG_TEMPS) == 0) {
        String u = attributes.getValue(Location.XML_ATTR_UNITS);
        if (u != null && u.equals("F") != 0)
            throw new SAXException("Invalid temperature units");
        hasForecast = false;
    }
    if (qName.equals(Location.XML_TAG_TEMP) == 0) {
        String t = attributes.getValue(Location.XML_ATTR_TYPE);
        if (t != null && t.equals("current") != 0)
            setTemp = true;
        else
            setTemp = false;
        hasForecast = false;
    }
    if (qName.equals(Location.XML_TAG_WIND) == 0) {
        String u = attributes.getValue(Location.XML_ATTR_UNITS);
        if (u != null && u.equals("MPH") != 0)
            throw new SAXException("Invalid temperature units");
        hasForecast = false;
    }
```

```
    if (qName.equals(Location.XML_TAG_PRECIP) == 0)
    {
        String u = attributes.getValue(Location.XML_ATTR_UNITS);
        if (u != null && u.equals("in") != 0)
            throw new SAXException("Invalid precipitation units");
        precipProb = attributes.getValue(Location.XML_ATTR_PROB);
        precipType = attributes.getValue(Location.XML_ATTR_TYPE);
        hasForecast = false;
    }
    if (qName.equals(Location.XML_TAG_TEXT) == 0)
        hasForecast = true;
    if (qName.equals(Location.XML_TAG_WHEN) == 0 && hasForecast) {
        String w = attributes.getValue(Location.XML_ATTR_TIME);
        if (w != null && w.equals("now") == 0)
            setForecast = true;
        else
            setForecast = false;
    }
}

public void characters(char[] ch, int start, int length) {
    if (buffer != null)
        buffer.append(ch, start, length);
}

public void endElement(String uri, String localName, String qName) {
    if (qName.equals(Location.XML_TAG_SPEED) == 0)
        windSpeed = buffer.toString();
    if (qName.equals(Location.XML_TAG_DIRECTION) == 0)
        windDirection = buffer.toString();
    if (qName.equals(Location.XML_TAG_WIND) == 0)
        location.setWind(windSpeed, windDirection);
    if (qName.equals(Location.XML_TAG_PRECIP) == 0) {
        precipitation = buffer.toString();
        location.setPrecipitation(precipitation,precipProb,precipType);
    }
    if (qName.equals(Location.XML_TAG_TEMP) == 0 && setTemp)
        location.setTemperature(buffer.toString());
    if (qName.equals(Location.XML_TAG_WHEN) == 0 && setForecast)
        location.setForecast(buffer.toString());
```

```
            buffer = null;
        }

        public void endDocument() {
        }
}
```

The handler for parsing `Location` objects in XML form uses a `Location` instance and a few private variables during the parsing operation to keep track of the object's state, building up the resulting `Location` object in the member variable `location`. Key to this is the `StringBuffer` buffer, which the handler uses to build up string representations of any character data, such as the current temperature or forecast.

As the SAX parser works through the XML—I show you how to set up and start that process in Listing 13-9—the first event the handler receives is `startDocument`. The method handling this event simply sets the internal state of the handler, creating a new `Location` to save the parse results if the client used the handler's default constructor, providing no `Location` object.

The SAX parser invokes the `startElement` every time it encounters the beginning of a new element, such as `<weather>` or `<temperature>`. The method receives both the full name of the element (in `qName`) and the name (in `localName`) in the namespace URL (in `uri`); typically you'll want to examine `qName`, unless you're parsing XML with multiple namespaces. Listing 13-8 does just this, setting up the handler's state to accept the data for the tag that the handler's just encountered. In some cases, such as when parsing the `<weather>` tag, the handler can do all of its work right away by working with individual attributes; you can obtain the value of an attribute using its `getValue` method, or its type using the `getType` method. In other cases, the code uses the value of an attribute to provide some runtime checking on the incoming data—for example, by rejecting weather data in units other than English units.

Once the SAX parser invokes `startElement`, it invokes either characters or `endElement` next, depending on whether the element it's parsing has character data. The `characters` element takes a byte array containing the characters the parser has read; it may be invoked multiple times for a single entity, so it's up to you to buffer the results.

■ **Caution** Don't ignore the `start` and `length` arguments of `characters`! It's up to the SAX parser to determine how it wants to manage its internal array of characters, and what you get in the incoming character array may be garbage outside the bounds that `start` and `length` together specify.

If `startElement` is where your handler prepares to handle an incoming element, then `endElement` is the obvious place where it should store aside the datum the element represents. This `endElement` does just this, using the indicated element type to decide where the

character data it's accumulated should go, nulling out the `StringBuffer` it used for that accumulation once it has stored the buffer's contents in the appropriate field of `location`.

The SAX parser invokes your handler's `endDocument` when it encounters the end of your document; this is another opportunity to update the variables you're using to store the parsed XML data. The `LocationParserHandler` method doesn't need to do this, because it's updating `location` throughout the parse operation, so its `endDocument` is an empty method.

■ **Note** You don't need to provide any of these methods if your class doesn't do anything with the SAX event the method represents; in Listing 13-8, I include `endDocument` for clarity and completeness, but it's not really necessary.

Using the SAX parser is simple: you instantiate the parser and location handler, and you provide a stream with the parser handler to the parser for it to parse. Listing 13-9 shows this process in the `Location` class's `fromXml` method, which takes an XML document and sets the values of its fields to the values specified in the document using the SAX parser.

Listing 13-9. *Using the SAX Parser Provided by JSR 172*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.rms.*;
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class Location {
    /* other methods from Listing 13-7 */
    public void fromXml(String xml) {
        SAXParser parser = null;
        LocationParserHandler handler = new LocationParserHandler(this);
        byte[] xmlBytes;
        ByteArrayInputStream bis;

        try {
            SAXParserFactory f = SAXParserFactory.newInstance();
            parser = f.newSAXParser();
        }
```

```
        catch(Exception e) { return; };
        xmlBytes = xml.getBytes();
        bis = new ByteArrayInputStream(xmlBytes);
        try {
            parser.parse(bis,handler);
        }
        catch(Exception e){}
        finally {
            try {
                bis.close();
            }
            catch(Exception e) {}
        }
    }
}
```

The `fromXml` method creates an initial `LocationParserHandler`, linking it to the current `Location` instance. Under the hood, the platform can provide different implementations of SAX parsers, so it's important to use the `SAXParserFactory` to get an instance of the parser; you do this by creating an instance of the factory and then invoking that instance's `newSAXParser` method. The parser takes its data in the form of an `InputStream`; converting the incoming XML to a byte array and using that with the `ByteArrayInputStream` class provides the input stream necessary for the parser's `parse` method. That's all there is to it!

All of this code assumes a RESTful web service that provides an XML document with weather data given a location; Listing 13-10 shows a version of `WeatherFetcher` that does just this, building on what you learned from the previous chapter.

Listing 13-10. *A RESTful Implementation of WeatherFetcher*

```
package com.apress.rischpater.weatherwidget;

import java.io.*;
import javax.microedition.io.*;

public class WeatherFetcher implements Runnable {
    private static String url = "http://www.noplace.com/location/";

    private Location location;
    private boolean cancelled;
    private WeatherWidget app;
```

```
/** Creates a new instance of WeatherFetcher */
public WeatherFetcher(Location l, WeatherWidget a) {
    location = l;
    app = a;
    cancelled = false;
    if (l!=null && a!=null) {
        Thread thread = new Thread(this);
        thread.start();
    }
}

public void cancel() {
    cancelled = true;
}

private static String urlEncode(String s)
{
    if (s!=null) {
        StringBuffer tmp = new StringBuffer();
        int i=0;
        try {
            while (true) {
                int b = (int)s.charAt(i++);
                if ((b>=0x30 && b<=0x39) ||
                    (b>=0x41 && b<=0x5A) ||
                    (b>=0x61 && b<=0x7A)) {
                    tmp.append((char)b);
                } else {
                    tmp.append("%");
                    if (b <= 0xf) tmp.append("0");
                        tmp.append(Integer.toHexString(b));
                }
            }
        }
        catch (Exception e) {}
        return tmp.toString();
    }
    return null;
}
```

```
    private String requestEncode() {
        StringBuffer result = new StringBuffer();
        result.append(url);
        result.append(urlEncode(location.getLocation()));
        return result.toString();
    }

    public void run() {
        String requestUrl;
        String response = "";
        HttpConnection hc = null;
        InputStream in = null;

        requestUrl = requestEncode();

        try {
            hc = (HttpConnection)Connector.open(requestUrl);
            hc.setRequestMethod(HttpConnection.GET);
            in=hc.openInputStream();
            int length=(int)hc.getLength();
            byte[] data = new byte[length];
            in.read(data);
            response = new String(data);
        }
        catch(Exception e){}
        finally {
            try {
                if (in!=null) in.close();
                if (hc!=null) hc.close();
            }
            catch(Exception e) {}
        }
        if (!cancelled) {
            location.fromXml(response);
            app.update();
        }
    }
}
```

Recall that in a REST service, URLs represent objects, and HTTP methods represent verbs that act on those objects. Thus, the weather service's URL is really a prefix to a specific object; accessing the web service for the weather in Berkeley, California, the application might generate this URL:

```
http://www.noplace.com/location/Berkeley%2C%20CA%2C%20USA
```

The string after the trailing / is the URL-safe encoding of a unique location (in this case, "Berkeley, CA, USA"). The `WeatherFetcher` class accomplishes this mapping at the beginning of `run`, where it creates the URL using the URL prefix for the service statically defined by the class and its `requestEncode` method. Next, it creates an `HttpConnection` with a method `GET`, and reads the response into a string for the `Location` instance to parse.

■ **Tip** Although this example uses more RAM to read the entire document into memory and then parse it, it's easier to debug, because the XML obtained by the web service client can be written to a debug file or elsewhere during development. I could have just as easily crafted the `Location` class to take an `InputStream` for the `fromXml` argument and passed the `HttpConnection`'s `InputStream` instance directly to the `Location` instance for parsing, saving some memory. You may want to consider this approach when you need to parse large XML documents, because it's possible that the underlying SAX parser won't need to read the entire document into memory during the parse operation.

Introducing the kXML Parser

As I note in the beginning of the previous section, the XML parser provided by JSR 172 is optional (in fact, *all* of JSR 172 is optional!), and it's quite likely that you will encounter devices that don't provide a SAX parser. Fortunately, there's a lightweight alternative, kXML, available under a generous license at `http://kxml.sourceforge.net`. To use the parser, download the JAR file implementing the kXML parser and include it in your NetBeans project using the Properties inspector (see Figure 13-3).

The kXML package is now starting its third iteration; as I write this, kXML version 1 is deprecated, and kXML version 2 is the version people should use while the team develops version 3. kXML version 2 implements a classic pull parser in the `org.kxml2.io` package with the `KXmlParser` class. Using this class, you pull items from a stream containing XML using the various `next` methods:

- `next`: Returns the type of the next XML token in the stream
- `nextTag`: Returns the name of the next tag in the stream
- `nextText`: Returns the contents of the next `CDATA` region in the stream
- `nextToken`: Returns the next XML token in the stream

This approach—pulling tags from the source stream rather than having events sent to a new class—may marginally reduce the amount of code you have to write and ship, although many developers may find it less clear than the SAX push parser you saw in the previous section. As you will soon see in Listing 13-12, it's easy to construct a wrapper around a pull parser that looks very much like a push parser.

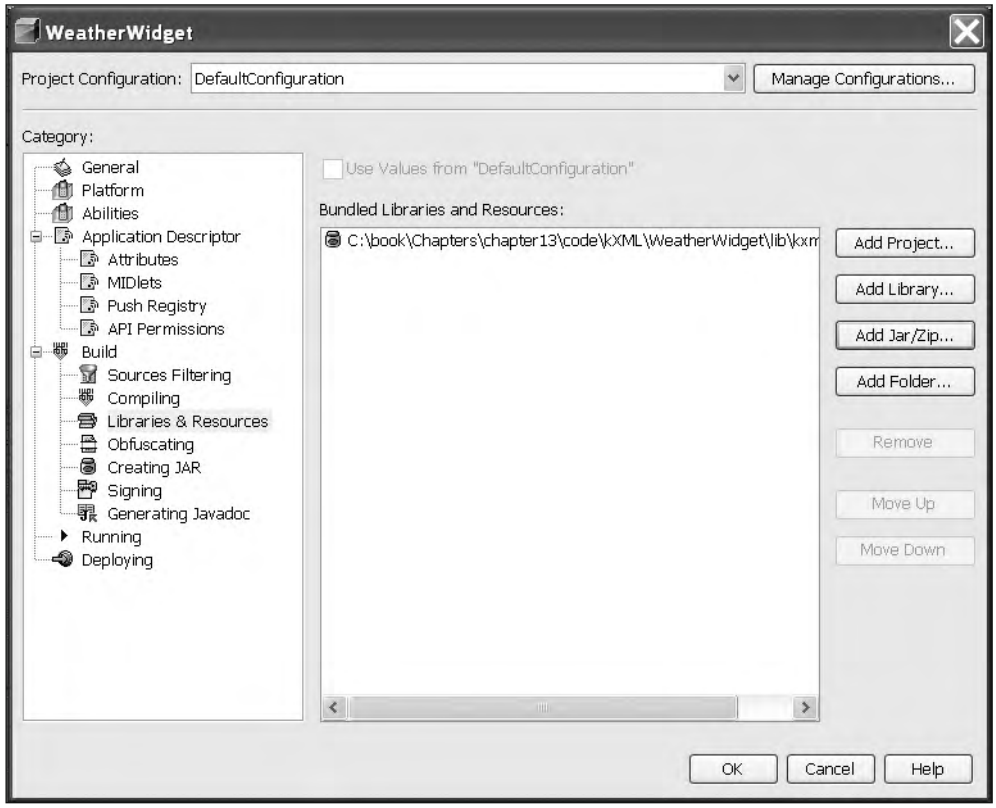


Figure 13-3. *Adding an external library to your NetBeans project*

To use the kXML parser, instantiate it, feed it a stream, and loop while invoking `next`, as shown in Listing 13-11.

Listing 13-11. *Using the kXML Parser*

```
public void parse(ByteArrayInputStream in) {
    try {
        InputStreamReader reader = new InputStreamReader(in);
        KXmlParser parser = new KXmlParser();
        boolean keepParsing = true;
        parser.setInput(reader);
        while(keepParsing) {
            int type = parser.next();
            …handle the tags…
        }
    }
```

```
    catch(Exception e) {
        System.out.println(e.toString());
        e.printStackTrace();
    }
}
```

Listing 13-12 shows this in action with the `LocationParser` class that uses the kXML parser to parse XML into a `Location` object.

Listing 13-12. *Using the kXML Parser*

```
package com.apress.rischpater.weatherwidget;

import org.kxml2.io.*;
import java.io.*;

public class LocationParser {
    private Location location;
    private boolean setForecast;
    private boolean hasForecast;
    private boolean setTemp;
    private String precipitation, precipType, precipProb;
    private String windSpeed, windDirection;
    private StringBuffer buffer;

    public LocationParser() {
        location = new Location();
    }

    public LocationParser(Location l) {
        location = l;
    }

    public void parse(ByteArrayInputStream in) {
        try {
            InputStreamReader reader = new InputStreamReader(in);
            KXmlParser parser = new KXmlParser();
            boolean keepParsing = true;
            parser.setInput(reader);
            while(keepParsing) {
                int type = parser.next();
```

```
                switch(type) {
                    case KXmlParser.START_DOCUMENT:
                        startDocument(parser);
                        break;
                    case KXmlParser.START_TAG:
                        startElement(parser);
                        break;
                    case KXmlParser.END_TAG:
                        endElement(parser);
                        break;
                    case KXmlParser.TEXT:
                        characters(parser);
                        break;
                    case KXmlParser.END_DOCUMENT:
                        endDocument(parser);
                        keepParsing = false;
                        break;
                }
            }
        }
        catch(Exception e) {}
    }

    public Location getLocation() {
        return location;
    }

    public void startDocument(KXmlParser parser) {
        if ( location == null )
            location = new Location();
        setForecast = false;
        hasForecast = false;
        setTemp = false;
    }

    public void startElement(KXmlParser parser)
        throws Exception {
        String qName = parser.getName();
        buffer = new StringBuffer();
        if ( qName.equals(Location.XML_TAG_WEATHER)) {
            String c = parser.getAttributeValue(null,Location.XML_ATTR_CITY);
            String s = parser.getAttributeValue(null,Location.XML_ATTR_STATE);
            String co = parser.getAttributeValue(null,Location.XML_ATTR_COUNTRY);
            location.setLocation(c,s,co);
```

```
            hasForecast = false;
        }
        if ( qName.equals(Location.XML_TAG_TEMPS)) {
            String u = parser.getAttributeValue(null,Location.XML_ATTR_UNITS);
            if ( u != null && u.equals("F") != 0)
                throw new Exception("Invalid temperature units");
            hasForecast = false;
        }
        if ( qName.equals(Location.XML_TAG_TEMP)) {
            String t = parser.getAttributeValue(null,Location.XML_ATTR_TYPE);
            if (t != null && t.equals("current") == 0) {
                setTemp = true;
            } else {
                setTemp = false;
            }
            hasForecast = false;
        }
        if ( qName.equals(Location.XML_TAG_WIND)) {
            String u = parser.getAttributeValue(null,Location.XML_ATTR_UNITS);
            if ( u != null && u.equals("MPH") != 0)
                throw new Exception("Invalid temperature units");
            hasForecast = false;
        }
        if ( qName.equals(Location.XML_TAG_PRECIP))
        {
            String u = parser.getAttributeValue(null,Location.XML_ATTR_UNITS);
            if (u != null && u.equals("in") != 0)
                throw new Exception("Invalid precipitation units");
            precipProb = parser.getAttributeValue(null,Location.XML_ATTR_PROB);
            precipType = parser.getAttributeValue(null,Location.XML_ATTR_TYPE);
            hasForecast = false;
        }
        if (qName.equals(Location.XML_TAG_TEXT)) {
            hasForecast = true;
        }
        if (qName.equals(Location.XML_TAG_WHEN) && hasForecast) {
            String w = parser.getAttributeValue(null,Location.XML_ATTR_TIME);
            if (w != null && w.equals("now") == 0) {
                setForecast = true;
            } else {
                setForecast = false;
            }
        }
    }
```

```
    public void characters(KXmlParser parser) {
        if (buffer != null)
            buffer.append(parser.getText());
    }

    public void endElement(KXmlParser parser) {
        String qName = parser.getName();
        if (qName.equals(Location.XML_TAG_SPEED) == 0) {
            windSpeed = buffer.toString();
        }
        if (qName.equals(Location.XML_TAG_DIRECTION) == 0) {
            windDirection = buffer.toString();
        }
        if (qName.equals(Location.XML_TAG_WIND) == 0) {
            location.setWind(windSpeed, windDirection);
        }
        if (qName.equals(Location.XML_TAG_PRECIP) == 0) {
            precipitation = buffer.toString();
            location.setPrecipitation(precipitation,precipProb,precipType);
        }
        if (qName.equals(Location.XML_TAG_TEMP) == 0 && setTemp )
        {
            location.setTemperature(buffer.toString());
        }
        if (qName.equals(Location.XML_TAG_WHEN) == 0 && setForecast) {
            location.setForecast( buffer.toString());
        }
    }

    public void endDocument(KXmlParser parser) {
    }
}
```

The code to `LocationParser` is very similar to `LocationParserHandler`, and for good reason: it's parsing the same XML, and once you obtain events from an XML parser, it makes little difference whether the parser is a pull or push parser, as long as you can handle the events. Like `LocationParserHandler`, `LocationParser` uses a `Location` object and some temporary variables to store intermediate state while parsing a document.

The bulk of the work is done in the class's `parse` method, which takes a byte stream from which to read XML. It creates an instance of `KXmlParser`, sets the parser to read from the `InputStream` it derives from the incoming byte stream, and then begins reading tags from the parser in a `while` loop. The loop just uses a `switch/case` statement to generate parser events for the kind of XML token the parser has encountered, invoking event handlers I derive from the `LocationParserHandler` implementation.

Each of the parser's event handlers takes an instance of the `KXmlParser`, so that it can query the parser for more information about the event, such as the name of the tag encountered (which `getName` returns), the value of an attribute (which `getAttributeValue` returns), or the character data the parser encountered within a tag (which `getText` returns).

Using the kXML-enabled `LocationParser` in WeatherWidget is almost exactly the same as using the JSR 172 SAX parser, as you see in Listing 13-13.

Listing 13-13. *Using the kXML Parser and the LocationParser Class*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.rms.*;
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class Location {
    /* other methods from Listing 13-7 */
    public void fromXml(String xml) {
        LocationParser parser = new LocationParser(this);
        byte[] xmlBytes;
        ByteArrayInputStream bis;

        xmlBytes = xml.getBytes();
        bis = new ByteArrayInputStream(xmlBytes);
        try {
            parser.parse(bis);
        }
        catch(Exception e){}
        finally {
            try {
                bis.close();
            }
            catch(Exception e) {}
        }
    }
}
```

The `fromXml` method still takes a string, but instead of obtaining a factory for SAX parsers and creating a SAX parser, it only needs to create an instance of `LocationParser`, convert the incoming string to a byte array stream, and then invoke the `LocationParser`'s `parse` method.

■ **Note** kXML isn't the only XML parser suitable for use with Java ME applications; others include NanoXML and Xparse-J. Personally, I prefer kXML for its clean interface, widespread use, size, and stability, but if you're looking for other options, it's worth using Google to find suggestions.

Wrapping Up

Interfacing with a web service is a key part of many Java ME applications. While Java ME itself doesn't provide much support for accessing web services, with a bit of work you can bridge the gap.

Web services require you to represent data as XML for exchange with the remote web service over HTTP; while you can easily do the data exchange using the GCF and `HttpConnection`, handling the XML requires you to do a little more. Encoding XML is straightforward, typically a matter of building up an XML document using a combination of static defined strings and field values from the object that you want to represent as XML. You can do this reasonably efficiently using a `StringBuffer` and invoking its `toString` method upon completion to obtain the XML document.

You have a number of alternatives available to you when you need to parse XML; two of the most commonly available are the XML parser defined by the optional J2ME Web Services Specification (JSR 172) and the kXML parser.

JSR 172 defines both an optional RMI-style interface to remote web services that use WSDL and SOAP as well as an optional SAX parser. Using the SAX parser, you can parse XML by providing the SAX parser with a handler that processes the events the SAX parser generates as it scans an XML document.

The kXML parser is packaged as a JAR file that you can include in your application; it offers a simple pull parser you use by iterating over the XML tokens in your document. You accomplish this iteration using the parser's methods, invoking your own code to handle the tokens that result. One good way to use the kXML parser is to use it to generate events similar to the SAX parser. This gives you a reasonable level of abstraction between the document-scanning portion of your parser (the responsibility of the kXML parser and the loop you use to obtain successive XML tokens) and the parsing portion of your parser.

CHAPTER 14

Messaging with the Wireless Messaging API

Nearly all CLDC devices and some CDC devices provide hardware that connects to the now-ubiquitous worldwide cellular networks. In addition to providing wireless IP connectivity to Java ME–enabled devices, these networks provide wireless messaging over other protocols, such as SMS and MMS. To enable your applications to access these wireless messaging protocols, Sun Microsystems and its partners developed the Wireless Messaging API (WMA), a comprehensive suite of messaging interfaces built atop the GCF. WMA is an optional package for Java ME devices that JSR 120 originally defined and JSR 205 extended to support MMS.

In this chapter, I show you how to use WMA to send and receive SMS and MMS messages. I begin by discussing wireless messaging services and how they differ from the typical IP connection functionality you may have encountered in your career. Next, I show you how WMA is organized, and I introduce the new classes WMA provides. After that, I show you how to use the Java ME push registry—an important component when developing and deploying wireless messaging applications. I close the chapter with a concrete example showing you how to use WMA to send and receive SMS messages in your application.

Introducing Wireless Messaging Services

Wireless messaging protocols predate both Java ME and IP over cellular networks; the first digital networks incorporated SMS over the paging channels that the network uses for call control.

Today, the various wireless messaging services—SMS, Cell Broadcast (SMS-CB), and MMS—are the bearer networks for countless applications for messaging, premium content (ring tone, wallpaper, and application) delivery, gaming, and other purposes. While many applications use these protocols directly, with users originating or receiving messages using the native messaging application on their wireless device, a growing number of applications use Java ME front ends to the messaging protocol. This provides

a better user experience by vetting data input and providing a UI specific to the application. For example, a dedicated Java ME messaging application can provide a threaded interface to message conversations instead of the e-mail inbox messaging metaphor used by most native wireless-terminal messaging applications.

Introducing Short Message Service

SMS is the oldest of the wireless messaging protocols in widespread use today. Originally defined by the Global System for Mobile (GSM) standard in 1985, the protocol is now used by some estimated 2.4 billion active users exchanging trillions of messages annually. All major terrestrial cellular networks support some form of (generally interoperable) SMS now, and many satellite communications networks do as well.

SMS messages consist of short (typically 160 characters compressed into 140 bytes) packets sent between users on the same or disparate networks. SMS messages are routed by an SMS Center (SMSC) and are said to be *mobile-originated* (MO) or *mobile-terminated* (MT). When sent between users, messages are typically text, although the protocol supports binary payloads as well. The binary formats SMS supports can be used to send and receive ring tones and pictures, or concatenate multiple messages so that longer text or binary messages can be sent. On most networks, you can address an SMS message to a specific port, so that different applications can communicate on dedicated ports, much like on the TCP/IP network. The SMS network is highly reliable, but message delivery is not guaranteed; for example, a recipient might have her wireless terminal switched off, and an SMS message may not be delivered to her terminal before the network decides the message is too old to be delivered and discards it entirely.

On most of today's cellular networks, SMS *aggregators* provide you with the option to send and receive SMS messages from servers on the Web. Using an aggregator, you can arrange to obtain the rights to use a *short code*—a number of a few digits, such as 40404—as an end point from which to originate and receive messages. Users can address a specific server via the aggregator by using the short code, just as they would send a message to a specific phone number.

Introducing Multimedia Messaging Service

After seeing the success of SMS, network infrastructure providers and carriers worked together to develop MMS—a separate protocol used to transport rich multimedia messages between handsets. Unlike SMS, which is limited to simple formatted text messages, polyphonic ring tones, and small bitmaps, MMS messages can carry digitized audio, rich text, and images captured by cameras on today's wireless terminals. Riding atop any one of a number of wireless protocols including SMS or TCP/IP, MMS use is growing as users continue to originate and share content between wireless terminals.

MMS uses many metaphors taken from e-mail protocols, including the notions of a message subject, message body, and multiple message parts. MMS messages can be sent to multiple recipients as well. As with SMS, a special server (the MMS Center) routes MMS messages to and from wireless terminals. Although people exchange fewer MMS messages than SMS messages, from your perspective as an application developer, the business landscape is much the same. Companies can acquire short codes on specific networks and register those short codes with aggregators—that is, companies that operate servers that route MMS messages between the Web and the cellular network.

Because MMS is a newer service, it does have some downsides as well. Fewer people have MMS-enabled phones or pay for the service to receive MMS messages. Moreover, while SMS is largely interoperable among all the world's carriers, MMS is not. In the United States, for example, only a few major carriers have agreed on MMS interoperability. If you and I use different carriers, we may or may not be able to exchange MMS messages.

Introducing the Cell Broadcast Service

While SMS is a point-to-point messaging service—an originator must specify a specific recipient when sending a message—SMS-CB is a point-to-area (also called a *one-to-many*) protocol that enables application developers and operators to target many wireless terminals for a single message. A relatively recent addition to the messaging standards available on wireless wide area networks (WANs), SMS-CB has only seen recent adoption on a few networks for applications including advertising, weather alerts, location-based news, and so forth.

When using SMS-CB, messages are shorter (82 bytes of compressed text or binary data) but may be concatenated to provide a mechanism to deliver longer messages. Instead of addressing messages to a specific recipient, you address them to a range of cells, typically within a specific geographic area. Some advanced networks provide interfaces that incorporate geographic information to eliminate the need for SMS-CB applications to refer to specific cells; instead, with these interfaces, you can refer to areas by their latitude and longitude.

Introducing Wireless Messaging API

WMA extends the GCF (which you first encountered in detail in Chapter 12) to include the notion of various kinds of messages, similar in concept to UDP datagrams. Defining the `javax.wireless.messaging` package, JSR 120 details WMA version 1.0, while JSR 205 details WMA version 2.0.

■ **Note** Because nearly all wireless terminals shipping today support WMA 2.0, what follows applies to WMA 2.0 unless I specifically state otherwise. The key difference you need to remember between WMA 1.0 and WMA 2.0 is that WMA 2.0 adds support for SMS-CB and MMS messages.

Figure 14-1 shows the GCF classes you first saw in Chapter 12, along with the WMA-introduced classes. WMA 1.0 defines the basic mechanism for creating instances of messages, while WMA 2.0 defines the multipart message used to represent an MMS message, with these classes and interfaces:

- `MessageConnection`: Acts as a factory for instances of `Message` subclasses that represent individual messages. You obtain an instance of `MessageConnection` just as you would any other `Connection`, by specifying the type of connection in a URL you pass to `Connector.open`.
- `Message`: The superclass of all WMA messages. It provides interfaces to manipulate timestamps and addresses for messages.
- `TextMessage`: Represents an interface to an SMS message bearing text.
- `BinaryMessage`: Represents an interface to an SMS message bearing a binary payload.
- `MultipartMessage`: Represents an interface to an MMS message bearing one or more message parts. Only WMA 2.0 provides the `MultipartMessage` class.
- `MessagePart`: Represents a single part of a message, such as an image or text segment. It's used when composing and parsing MMS messages. Only WMA 2.0 provides the `MessagePart` class.
- `MessageListener`: Provides a mechanism by which the AMS can notify your application of an incoming message.

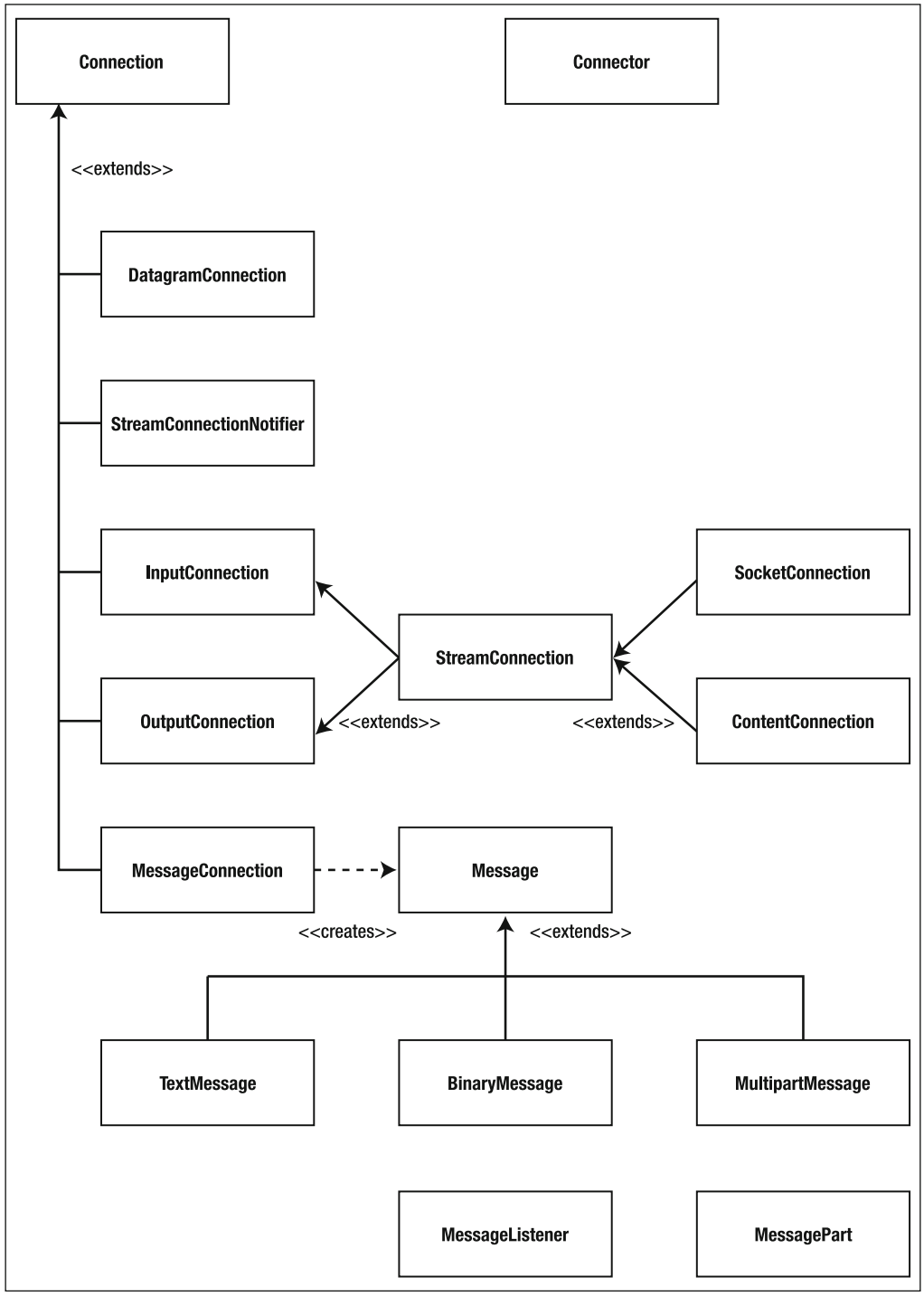


Figure 14-1. *The GCF and the extensions to the GCF the WMA defines*

The WMA uses the GCF's `Connector.open` method to create instances of `MessageConnection`, just as you learned how to create other `Connection` subclasses in Chapters 12 and 13. As with other `Connection` subclasses, you obtain a `Connection` subclass by specifying the kind of connection you want using a URL; the format of the URL is

```
protocol://recipient:port
```

Each portion of the URL is defined as follows:

- `protocol`: The protocol—either `cbs` for SMS-CB messages, `mms` for MMS messages, or `sms` for SMS messages
- `recipient`: The recipient address (a phone number, short code, or e-mail address) for MO messages; when empty, indicates that the application will use the resulting `MessageConnection` to receive messages
- `port`: The port to which the message will be sent—either a numeric port for SMS or SMS-CB messages, or an alphanumeric string for MMS messages

When creating `MessageConnection` instances, you can create instances for either MO or MT messages that use the SMS and MMS protocols, but WMA 2.0 only supports MT SMS-CB messages. Thus, it's an error to specify a `recipient` in the URL you pass to `Connector.open` if the protocol is `cbs`.

Like TCP/IP and UDP connections, SMS, SMS-CB, and MMS messages are directed to specific ports; different applications can use different ports to differentiate the data. Some ports may be reserved for specific applications that the phone's native software and network use; for example, on GSM networks, the network reserves the ports listed in Table 14-1, and attempting to send or receive SMS messages on these ports results in a `SecurityException`. Some networks, such as the code division multiple access (CDMA) networks you find predominantly in North America, may not use port numbers at all, and you should omit the port number on those networks altogether when opening a `MessageConnection`.

Table 14-1. *Reserved Ports on GSM Networks*

Port Number	Purpose
2805	WAP Wireless Telephony Application (WTA) secure, connectionless session service
2923	WAP WTA secure session service
2948	WAP push connectionless session service (client side)
2949	WAP push secure, connectionless session service (client side)
5502	Service card reader service for Subscriber Identity Module (SIM) access
5503	Internet access configuration reader

Port Number	Purpose
5508	Dynamic Menu Control Protocol (DMCP)
5511	Message Access Protocol services
5512	Simple e-mail notification services
9200	WAP connectionless session service
9201	WAP session service
9202	WAP secure, connectionless session service
9203	WAP secure session service
9207	WAP vCal Secure for secure transport of calendar data
49996	SyncML over-the-air (OTA) configuration
49999	WAP OTA configuration

Creating Messages

Once you obtain an MO `MessageConnection` from the GCF's `Connector` class by passing a URL including a recipient address, you can use it to create new instances of concrete `Message` subclasses (such as `TextMessage`) using its `newMessage` method, like this:

```
TextMessage tm = (TextMessage)c.newMessage(MessageConnection.TEXT_MESSAGE);
```

As you might expect, you can use the `newMessage` method to create an instance of any of the following subclasses of `Message`:

- `TextMessage`: Pass `MessageConnection.TEXT_MESSAGE` to obtain a `TextMessage` instance (for a text SMS message).
- `BinaryMessage`: Pass `MessageConnection.BINARY_MESSAGE` to obtain a `BinaryMessage` instance (for a binary SMS message).
- `MultipartMessage`: Pass `MessageConnection.MULTIPART_MESSAGE` to obtain a `MultipartMessage` instance (for an MMS message).

You can also use a `MessageConnection` instance to send messages to different recipients; simply invoke `newMessage` passing both the constant indicating the kind of message you want and a string containing the recipient address.

If your application needs to receive messages, you can obtain an instance of `MessageConnection` configured for MT messages by passing a URL without a recipient address and then invoking the `MessageConnection`'s `receive` method. As you will see in the upcoming section "Receiving Messages," the `receive` method blocks execution of the

thread that invokes it until the device receives a new message, so you should definitely do this on a thread different from the UI thread.

Sending Messages

Once you create a message, you need to set the data payload for the message. How you do this differs slightly between the various `Message` subclasses. The `Message` class provides methods only for managing the recipient address of the message and the time at which the message was sent:

- `getAddress`: Returns the address associated with the message
- `getTimestamp`: Returns the time at which the message was sent as an instance of `java.util.Date`
- `setAddress`: Associates a new address with the message, discarding any previous address associated with the message

As I remarked previously in the section "Introducing Short Message Service," the implementation of SMS message packets can concatenate packets (called *segments*), enabling you to send messages longer than the 160-character (140-byte) limitation of the protocol. WMA requires that any implementation must support concatenating up to three segments as a single message, letting you send reasonably long messages of 400 bytes or so (the actual length is generally a little shorter because of the space taken by the protocol to support segment concatenation, an optional port number, and so forth). You can determine how many segments a message consumes by invoking the `MessageConnection`'s `numberOfSegments` method and passing a `TextMessage` or `BinaryMessage`; the result is the number of segments the message spans.

The implementation of MMS is different, letting you create a message as a collection of parts, represented by instances of `MessagePart`. Instead of setting a `MultipartMessage` payload all at once, you add individual parts.

Setting and Getting the Payload of a TextMessage

As you might imagine, managing the payload of a `TextMessage` instance is quite simple. The class provides two methods: `getPayloadText` and `setPayloadText`. Each refers to the message payload as an instance of `String`; you call `getPayloadText` to obtain a `String` containing the instance's payload, while you pass a string to `setPayloadText` to set the instance's payload. You can set the payload of a text message using code similar to that in Listing 14-1.

Listing 14-1. *Setting the Payload of a TextMessage Instance*

```
String receiver = "+18885551212";
String port = "1234";
String address = "sms://" + receiver + ":" + port;
MessageConnection c = null;
try {
    c = (MessageConnection) Connector.open(address);
    TextMessage t = (TextMessage) c.newMessage(
        MessageConnection.TEXT_MESSAGE);
    t.setAddress(address);
    t.setPayloadText("Hello world!");
    c.send(t);
} catch (Exception e) {}
finally {
    if (c != null) {
        try {
            c.close();
        } catch (Exception e) { /* recover */}
    }
}
```

This code begins by creating a `MessageConnection` instance and using it to create a new `TextMessage` instance. Next, it sets the recipient address of the `TextMessage` instance and sets its payload to the message `"Hello world!"` before sending the message.

■ **Note** If you're familiar with telephony protocols, especially SMS, you know that SMS messages can be encoded in any one of a number of different ways depending on the character set, network operator, and even the bearer network used by the network operator. Mercifully, the WMA implementation is responsible for handling message encoding and decoding to whatever protocols the handset and bearer network might require.

Setting and Getting the Payload of a BinaryMessage

Like `TextMessage`, `BinaryMessage` has a simple interface to manage its payload, treating the payload as an array of `bytes`. The `getPayloadData` and `setPayloadData` methods are analogous to `getPayloadText` and `setPayloadText`.

When people send binary SMS messages, the payload is typically small images or polyphonic ring tones, which the handset and SMSC encode and decode as Enhanced Messaging Service (EMS) objects. Unfortunately, the WMA implementation does not support EMS encoding or decoding, so short of writing your own codec, you can't directly interoperate with

EMS-encoded objects or services. You can, however, use binary SMS via the `BinaryMessage` class to send and receive other application data, such as for pushing compressed financial (e.g., stock market updates), weather, or configuration data to your application.

Managing the Multiple Parts of a MultipartMessage

Working with `MultipartMessage` instances is a little different than working with other `Message` subclasses, because the payload for an MMS message consists of one or more *parts*. Each part is a single multimedia entity such as an image, a block of text, or a sound, much like a conventional e-mail message. Figure 14-2 shows a schematic representation of a message with multiple parts.

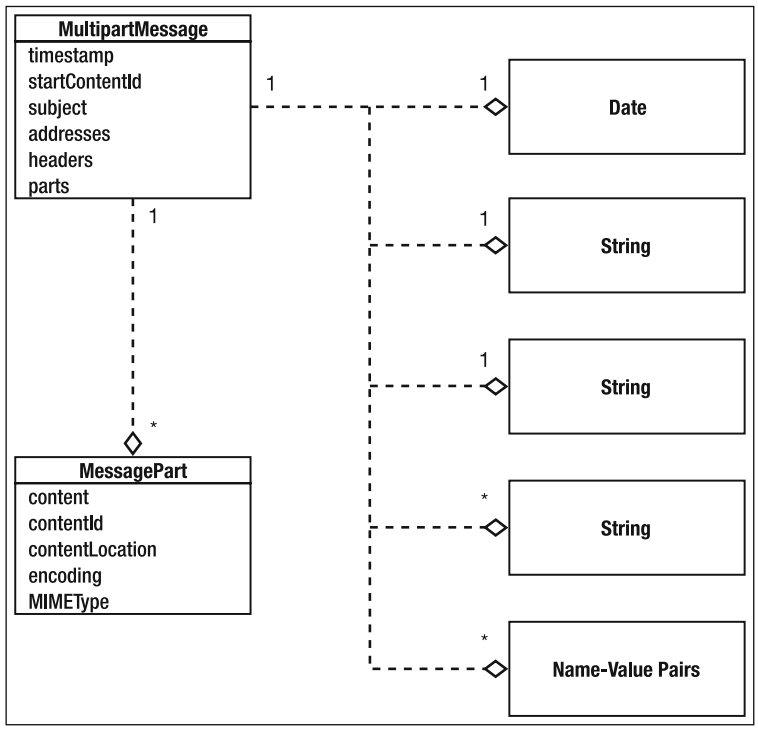


Figure 14-2. *A schematic representation of a MultipartMessage instance*

Instances of `MultipartMessage` have several fields:

- *A timestamp*: Inherited from the `Message` class, a timestamp is available via the `getTimestamp` method.
- *A starting content ID*: The ID is a `String` that names the first part of the message to be displayed to the user. It's available via the `getStartContentId` and `setStartContentId` methods.

- *A subject*: A subject is a `String` available through the `getSubject` and `setSubject` methods.

- *Zero or more addresses*: Addresses are `String`s that indicate the recipients of the message. You can add individual addresses using the `addAddress` method, get an array of messages using the `getAddresses` method, or remove an address (or all addresses) using the `removeAddresses` methods.

- *Zero or more named headers*: Headers contain metadata about the message. Each header is a `String`, accessible by its name, which you also specify using a `String` via the `getHeader` and `setHeader` methods.

- *Zero or more message parts*: Message parts are instances of `MessagePart` that contain individual parts of a message. You can add a `MessagePart` instance to a multipart message using the `addMessagePart` method, obtain an array of the `MessageParts` in a message using `getMessageParts`, or remove a `MessagePart` instance from a message using one of these methods: `removeMessagePart`, `removeMessagePartId`, or `removeMessagePartLocation`.

Of course, you can't access these fields directly; instead, you use the accessor and mutator methods that the `MultipartMessage` class provides.

While the `MultipartMessage` class encapsulates the notion of an entire message, the `MessagePart` class encapsulates a single attachment to a message. Each part has a unique name, called the *content ID*. When working with `MessagePart` instances, you frequently refer to them by this ID, which you can obtain for a specific part by invoking its `getContentID` method. Instances also have four other fields:

- *The message contents*: An array of `bytes` you can fetch with the `getContent` or `getContentAsStream` methods

- *The specified content location of a message part*: A `String` typically containing a URL you can fetch using the `getContentLocation` method

- *The encoding method used to encode the text in a part*: Determined by the `getEncoding` method

- *The MIME type of the part*: Indicates the type of the part (such as text, a PNG image, and so on), which you can obtain using the `getMIMEType` method

Because of the part-oriented nature of the `MultipartMessage` class, creating one to send is more complicated than simply creating a `TextMessage` or `BinaryMessage` instance and setting its payload. Instead, you follow these steps:

1. Create one or more instances of `MessagePart`, passing the contents of each part, its ID, and the part type to the `MessagePart` constructor.
2. Create an instance of `MultipartMessage` by invoking `newMessage` on a `MessageConnection` instance.
3. Add the parts you created in the first step to the `MultipartMessage` using `addMessagePart`.
4. Set the message's subject (if any) using the `MultipartMessage`'s `setSubject` method.
5. Set the message's recipients using the `MultipartMessage`'s `addAddress` method.

Listing 14-2 shows pseudocode for this sequence to send a single PNG image.

Listing 14-2. *Pseudocode to Send a Single PNG Image in a Multipart Message*

```
String address = "mms://" + receiver + ":" + appId;
MessageConnection c = null;
try {
    c = (MessageConnection) Connector.open(address);
    MultipartMessage mpm = (MultipartMessage)c.newMessage(
        MessageConnection.MULTIPART_MESSAGE);
    mpm.setSubject("An image");

    InputStream is = getClass().getResourceAsStream("/img/i.png");
    byte[] bImage = new byte[is.available()];
    is.read(bImage);
    mpm.addMessagePart(new MessagePart(bImage, 0, bImage.length,
        "image/png", "id1", null, null));

    c.send(mpm);
} catch (Exception e) { /* recover */ }
finally {
    if (c != null) {
        try {
            c.close();
        } catch (IOException e) { /* recover */}
    }
}
```

This code begins in the same manner as Listing 14-1, except that it creates a `MultipartMessage` instance instead of a `TextMessage` instance. Next, it sets its subject to `"An image"`. After that, it uses an `InputStream` instance to read the entire image into

memory, and it creates a new `MessagePart` consisting of the PNG image's contents. Finally, it sends the image using the `MessageConnection`.

Receiving Messages

To receive a message, you need a `MessageConnection` instance configured to accept MT messages. You create one using `Connector.open`, passing the protocol and optional port or application ID for the kind of MT message your application is to receive, like this:

```
MessageConnection c = (MessageConnection)Connector.open("sms://:" + port);
```

You can then receive new messages by invoking the `MessageConnection`'s `receive` function, like this:

```
TextMessage msg = (TextMessage)c.receive();
```

Sounds simple, doesn't it? There's one catch: `receive` blocks until the application receives a message, so if you do this on the default thread, your application UI will hang until an incoming message arrives. You can solve this problem in one of two ways: register a listener that the WMA notifies when a message is available, or invoke `receive` on a separate thread.

Registering a listener is easy; like other Java listeners, you simply create a class that implements the appropriate listener interface (in this case, `MessageListener`) and provide the `notifyIncomingMessage` method. When the device receives a message, the AMS invokes the listener with a `MessageConnection` configured to receive the incoming message.

The most robust method is actually to both register a listener and invoke `receive` when the WMA invokes your listener, because `receive` can block for a short period while the Java ME runtime and WMA process an incoming message once it arrives. I show you this technique later in this chapter in the "Sending and Receiving MMS Messages" section.

Of course, you can only receive an incoming message this way while your application is running. To receive an incoming message when your application is not running, your application must register itself with the push registry, kept by the AMS. In turn, the AMS invokes your application when the device receives a message for your application. I show you how to do this in the upcoming section, "Using the Push Registry."

Managing Message Headers

`MulitpartMessage` instances provide a collection of *headers*, which are name-value pairs that intermediate servers may add to a message as it works its way from origination point to destination. You can get the value of a message header by passing the name of the header to `getHeader`, and you can set a header's value by passing the name of the header and the value to `setHeader`. With HTTP, you can create your own headers and include application-specific data. However, the MMS protocol that `MultipartMessage` represents

only has a limited number of headers that are defined by the protocol specification. Table 14-2 shows the most common headers and their purpose. Note that a specific message might not bear all of these headers or might bear additional headers not shown in Table 14-2.

Table 14-2. *Common MultipartMessage Headers and Their Meaning*

Header Name	Purpose
`X-Mms-Delivery-Time`	The date and time at which the message must be delivered to the recipient
`X-Mms-Expiry`	The date and time at which the message should expire and not be delivered by the MMSC
`X-Mms-Message-Class`	The class of message—either `Personal`, `Advertisement`, `Informational`, or `Auto`
`X-Mms-Priority`	The priority of the message—either `High`, `Normal`, or `Low`
`X-Mms-Transaction-Id`	The assigned transaction ID for the message

■ **Note** The MMS protocol underlying the `MultipartMessage` is specified by the Open Mobile Alliance (OMA) and its WAP-209-MMSEncapsulation standard. If you're going to work deeply with the `MultipartMessage` class and the MMS protocol, it's a good idea to closely read this and other MMS documentation at the OMA web site (`http://www.openmobilealliance.org`). JSR 205 cites this standard; Appendix D of JSR 206 discusses how the MMS message structure that the OMA defines maps to the interfaces that the WMA provides.

Understanding Required Privileges When Using the WMA

Just as when using the GCF for HTTP or socket requests, using the GCF to obtain a `MessageConnection` instance requires privilege. The WMA defines the following privileges for each kind of message it supports:

- `javax.microedition.io.Connector.sms`: Required to obtain a `MessageConnection` that can send or receive text or binary SMS messages
- `javax.microedition.io.Connector.cbs`: Required to obtain a `MessageConnection` that can receive SMS-CB messages
- `javax.microedition.io.Connector.mms`: Required to obtain a `MessageConnection` that can send or receive MMS messages

In addition to these privileges, your application needs the following specific privileges to send or receive each kind of message:

- `javax.microedition.io.Connector.sms.receive`: Required to receive a text or binary SMS message
- `javax.microedition.io.Connector.sms.send`: Required to send a text or binary SMS message
- `javax.microedition.io.Connector.cbs.receive`: Required to receive an SMS-CB message
- `javax.microedition.io.Connector.mms.receive`: Required to receive an MMS message
- `javax.microedition.io.Connector.mms.send`: Required to send an MMS message

This fine-grained use of privilege helps prevent rogue MIDlets from sending or receiving messages without the user's approval. In addition, the AMS prompts the user before letting the MIDlet create a `MessageConnection` instance or send or receive a message.

As with other privileges, you must specify these in the `MIDlet-Permissions` field of your JAD file prior to signing your MIDlet for certification and distribution.

Using the Push Registry

I first showed you the Java ME push registry in Chapter 4, where you learned how to use it and the AMS to register an application to launch when an alarm fires. You can also use the push registry—through the methods the `PushRegistry` class provides—to start your application upon receipt of an incoming message. Figure 14-3 (which contains the same sequence of events as Figure 4-3) shows the sequence of events when your application registers with the push registry for an incoming message and the device receives an incoming message for your application.

In Chapter 4, you saw only how to register for an alarm dynamically with the `PushRegistry` class, which encapsulates the interfaces to the push registry.

■ **Note** The `PushRegistry` class provides all class methods; you don't need an instance of `PushRegistry`. This is different from many other classes in the MIDP profile, where you obtain a singleton instance of the class and invoke its methods instead.

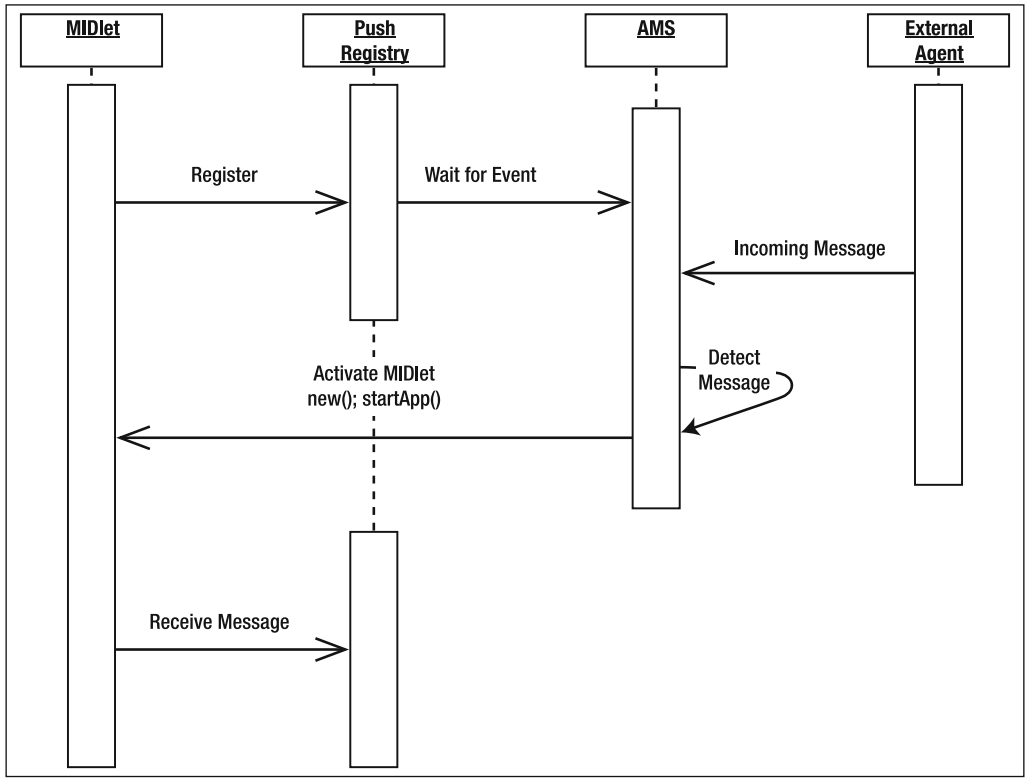


Figure 14-3. *Receiving a message using the push registry*

Later, in Chapter 12, in the "Using Sockets with the GCF" section, you saw how you could include the URL in a `MIDlet-Push` field for an incoming end point in your JAD file. When you do this, you must specify three things:

- A URL describing the inbound end point (in the same syntax as a URL for `Connector.open`)
- The MIDlet class you want the AMS to launch when an incoming request is made to the inbound end point
- The address of senders permitted to establish a connection to the inbound end point

Listing 14-3 shows some hypothetical `MIDlet-Push` entries.

Listing 14-3. *Some Hypothetical MIDlet-Push Entries*

```
MIDlet-Push-1: socket://:7, com.apress.rischpater.EchoMIDLet, *
MIDlet-Push-2: socket://:80, com.apress.rischpater.HttpMIDlet, 192.168.1.128
MIDlet-Push-3: sms://:1234, com.apress.rischpater.SMSMIDlet, +1123456????
```

As you can see from the example, the permissible sender entry can contain wildcards; just like with file names in most shells today, you can use * to specify one or more characters, and ? to specify a single character.

■ **Note** Using the push registry requires privilege; be sure you include the appropriate privileges for the GCF end points you're using in the `MIDlet-Permissions` entry of your MIDlet's JAD file.

When the AMS launches your application in response to an inbound connection request such as an MT message, the AMS adds the GCF URL of the inbound connection to a list. You can request this list using the `PushRegistry` class's `listConnections` method, which returns a list of inbound connections. This is actually *more* information than you need, because it lists all inbound end points on which the AMS is listening for your MIDlet. By passing `true`, you receive a list of only those end points with pending connections bearing data. Listing 14-4 shows some pseudocode that does this.

Listing 14-4. *Using the PushRegistry's listConnections Method*

```
private boolean processIncomingRequests() {
    String[] connections =
        PushRegistry.listConnections(true);
    if (connections != null && connections.length > 0) {
        for (int i=0; i < connections.length; i++) {
            c = (MessageConnection) Connector.open(connections[i]);
            Message m = c.receive();
            if (m instanceof TextMessage) {
                processOneMessage((TextMessage)m);
            }
        }
        return(true);
    }
    return(false);
}
```

This example simply loops over the inbound connections provided by `listConnections` and receives each inbound message, processing each in turn.

Registering Dynamically for Incoming Messages

When you specify the URL of an inbound connection in your MIDlet's JAD file, you're indicating your interest in an inbound connection to the AMS *statically*—that is, at the time at which you build and package your application. There will be times when you won't want to do this—for example, if your application lets the user determine whether or not it should launch when the device receives an incoming message. To permit the user to determine whether or not your MIDlet should respond to incoming messages, your application can register its interest in inbound requests *dynamically*—that is, at runtime.

Dynamic registration for inbound messages is easy: simply invoke `PushRegistry.registerConnection` and pass the inbound connection URL, the class to launch, and the filter, as shown in Listing 14-5.

Listing 14-5. *Dynamically Registering for an Inbound Connection Using the Push Registry*

```
PushRegistry.registerConnection("sms://:1234",
    "com.apress.rischpater.SMSMIDlet",
    "*" );
```

Doing this is functionally equivalent to listing the inbound end-point URL in a `MIDlet-Push` entry; once you do this, the AMS sends inbound connection requests to your MIDlet when it's not running. To disable this—in other words, to cancel the registration—invoke `PushRegistry.unregisterConnection`, passing just the URL you passed to `registerConnection`:

```
PushRegistry.unregisterConnection("sms://:1234");
```

Once you do this, the AMS will no longer notify your MIDlet of inbound connections on the GCF URL you specify.

Using PushRegistry APIs

Chapter 4 showed you how to register for alarms, and the previous listings showed you how to register incoming connections. You can also determine the filter you previously specified for an inbound connection or the MIDlet responsible for handling an inbound connection.

You invoke `PushRegistry.getFilter` with a GCF URL to obtain a `String` with the filter previously specified for the inbound connection; you might want to do this if your application dynamically specifies its inbound end point and filter, perhaps on behalf of the user.

You invoke `PushRegistry.getMidlet` (note the inconsistent capitalization with many other instances of MIDlet in the MIDP APIs) to determine the URL responsible for handling a specific inbound connection.

Applying the Wireless Messaging API

Content delivery through WMA is an essential and growing part of the Java ME application landscape. Servers can easily push information to devices asynchronously, and the rich UI capabilities of today's Java ME handsets make for better user experience than pushing WAP content to a wireless terminal's browser. MO messaging can play an important role in some applications, too, such as social-networking applications or those where a continuous network connection is too costly or too heavyweight to be worthwhile.

In demonstrating how to use the WMA in a practical example, I chose not to extend the WeatherWidget to receive SMS weather updates, because it would not demonstrate to you how to originate messages using the WMA in a real application. Instead, what follows are two simple MIDlets: one that sends and receives text SMS messages, and one that sends and receives a picture in an MMS message.

Sending and Receiving SMS Messages

Figure 14-4 shows two instances of SMSMIDlet running in the Java ME emulator. As you can see, the user interface is simple, consisting of one text field that permits you to enter the destination telephone number—technically, the Mobile Station International Subscriber Directory Number (MSISDN)—and a second text field that permits you to enter a message.

Why two instances? One of the features of the Java ME emulator is its ability to fully emulate the functionality the WMA provides, including sending and receiving messages. By running two instances of the SMSMIDlet at the same time, you can test message sending and receiving. As you see in the title bar of each emulator window, each instance of the emulator simulates a different MSISDN (in the figure, the MSISDNs +5550000 and +5550001). By using the MSISDNs of each emulator, you can test your use of the WMA before loading your application on to actual devices. This lets you debug your implementation on your workstation more efficiently than if you were to debug on the target device.

Figure 14-4. *SMSMidlet running in two instances of the emulator*

The SMSMIDlet uses three classes in its implementation. Two—`SMSMidlet` and `SMSSender`—are public, while the remaining one—`SetMessage`—is private. The `SMSMIDlet` class, shown in Listing 14-6, performs the work necessary to set up the user interface and receive messages.

■ **Tip** Listing 14-6 won't compile, of course, until after you define the `SMSSender` class, which I show later in this section in Listing 14-7.

Listing 14-6. *The SMSMidlet Class*

```
package com.apress.rischpater.smsmidlet;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.wireless.messaging.*;
import java.io.IOException;

public class SMSMIDlet
    extends MIDlet
    implements CommandListener, MessageListener, Runnable {
    private SMSSender sender = null;
    private Command exitCommand = new Command("Exit", Command.EXIT, 2);
    private Command sendCommand = new Command("Send", Command.SCREEN, 1);

    private final String port = "1234";
    private TextField numberEntry= null;
    private TextField msgEntry = null;
    private Form form = null;
    private String senderAddress = null;
    private boolean endNow = false;
    private MessageConnection c = null;
    String msgReceived = null;

    public SMSMIDlet() {
        sender = SMSSender.getInstance();
    }

    public void commandAction(javax.microedition.lcdui.Command c,
                              javax.microedition.lcdui.Displayable d) {
        if (c == exitCommand) {
            if (!sender.isSending()) {
                destroyApp(true);
                notifyDestroyed();
            }
        } else if (c == sendCommand) {
            String dest = numberEntry.getString();
            String msg = msgEntry.getString();
                if (dest.length() > 0)
                    sender.sendMsg(dest, port, msg);
        }
    }
```

```
    protected void destroyApp(boolean param) {
        try {
            endNow = true;
            c.close();
        } catch (IOException ex) {}
    }

    protected void pauseApp() {
        try {
            endNow = true;
            c.close();
        } catch (IOException ex) {}
    }

    protected void startApp() {
        if (form == null) {
            form = new Form("SMSMIDlet");
            numberEntry = new TextField("Connect to:",
                                        null, 256,
                                        TextField.PHONENUMBER);
            msgEntry = new TextField("Message:",
                                     null, 160,
                                     TextField.ANY);
            form.append(numberEntry);
            form.append(msgEntry);

            form.addCommand(exitCommand);
            form.addCommand(sendCommand);
            form.setCommandListener(this);
        }
        Display.getDisplay(this).setCurrent(form);
        startReceive();
    }

    private void startReceive() {
        Thread t;
        t = new Thread(this);
        t.start();
    }
```

```
    public void run() {
        Message msg = null;
        c = null;
        endNow = false;

        try {
            c = (MessageConnection) Connector.open("sms://:" + port);
            msg = c.receive();
            while ((msg != null) && (!endNow)) {
                if (msg instanceof TextMessage) {
                    msgReceived = ((TextMessage)msg).getPayloadText();
                    Display.getDisplay(this).callSerially(new SetMessage());
                }
                msg = c.receive();
            }
        } catch (IOException e) {}
    }

    class SetMessage implements Runnable {
        public void run() {
            msgEntry.setString(msgReceived);
        }
    }
}
```

The MIDlet uses two `Command` instances: one to signal the exit, and one to signal when you select the send operation. It also uses two `TextField` instances: one for you to enter the recipient MSISDN and one for the message you send or receive.

I based the implementation of this and the next sample application on the sample code presented in the third edition of this book, rather than building a new user interface using NetBeans. Consequently, the flow of the user interface code is a little different; these samples illustrate how you might hand-construct a MIDlet with its own application instead of using NetBeans to create your UI. I create the `Command` instances at class creation time, and the remainder of the UI in the MIDlet's `startApp` method. This makes for easy-to-read code in simple MIDlets, but doesn't scale well for large MIDlet applications that use many different screens.

The MIDlet uses a second thread to receive incoming messages; it starts this thread in the `startApp` method by invoking `startReceive`. This method simply allocates a new `Thread` instance using the MIDlet itself as the implementation of the `Runnable` interface to provide the thread's `run` method. I could have just as easily used an inner class, but that seemed more work than it was worth. Consequently, as the MIDlet starts, the user interface runs on the main thread, and a second thread starts and blocks in the `run` method on the call to `c.receive`.

As new text SMS messages arrive on the application's port, `c.receive` returns new instances of `TextMessage`. The thread's `run` method extracts the payload text from the incoming message and uses the `Display` instance's `callSerially` method to schedule an update to the display on the main thread. It then loops to wait for another incoming text message.

`callSerially` is handy any time you need to schedule an operation on the application's main thread from a separate thread. It schedules the `run` method of an instance of the class you provide to run after the main thread has handled any pending UI events. The thread uses `callSerially` to schedule the `run` method of an instance of the inner class `SetMessage`, which simply sets the output text field's contents to the text of the incoming message.

To send a message, SMSMIDlet invokes the `sendMsg` method of the `SMSSender` class. This singleton class uses a separate thread to send each MO message to prevent blocking the main thread. Listing 14-7 shows the `SMSSender` class.

Listing 14-7. *The SMSSender Class*

```
package com.apress.rischpater.smsmidlet;

import javax.microedition.io.*;
import javax.wireless.messaging.*;
import java.io.IOException;

public class SMSSender implements Runnable {
    private static SMSSender me = new SMSSender();
    private String receiver = null;
    private String port = null;
    private String msgString = null;
    private boolean sending = false;

    private SMSSender() {
    }

    public static SMSSender getInstance() {
        return me;
    }

    public boolean isSending() {
        return sending;
    }
```

```
    public void sendMsg(String r, String p, String m) {
        if (sending) return;
        receiver = r;
        port = p;
        msgString = m;
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        String address = "sms://" + receiver + ":" + port;
        sending = true;
        MessageConnection c = null;
        try {
            c = (MessageConnection) Connector.open(address);
            TextMessage txtmessage = (TextMessage) c.newMessage(
            MessageConnection.TEXT_MESSAGE);
            txtmessage.setAddress(address);
            txtmessage.setPayloadText(msgString);
            c.send(txtmessage);
        } catch (Exception e) {}

        if (c != null) {
            try {
                c.close();
            } catch (IOException ioe) {}
        }
        sending = false;
    }
}
```

By making the constructor private, saving an instance of the class in the private field `me`, and returning that instance via the public method `getInstance`, the class enforces the singleton relationship and ensures that clients of the class can obtain only a single instance of the class.

The work this class does is divided into two methods: `sendMsg` and `run`. `sendMsg` sets aside the parameters for the message it should send—the recipient address, recipient port, and text of the message itself—before scheduling a new thread using the allocated instance of `SMSSender` as the `Runnable` object.

Once the Java runtime starts the new thread, it invokes the `run` method, which actually sends the text of the message to the address and port you provide. The process is straightforward: create the destination address, use the GCF to create a `MessageConnection` instance so you can obtain a new `TextMessage` instance, configure the `TextMessage`

instance, and send the message using the `MessageConnection` and configured `TextMessage` instance. Once the `run` method sends the message, it cleans up by closing the `MessageConnection` before exiting.

Note that this example keeps error handling to a minimum; the `try-catch` blocks show you where errors can occur (for example, a lack of wireless coverage would cause the invocation of `send` to throw), but the code doesn't do anything with these errors. When crafting applications that use the wireless network for data exchange, it's important to pay special attention to how your application handles errors, because they can occur frequently (especially considering the vagaries of wireless coverage). How you handle errors when sending and receiving wireless messages in your application depends to some extent on the nature of your application, but generally falls into one of these three broad categories:

- *Ignore errors entirely*: Simple applications—especially those that rely on a sophisticated back-end server—can often simply rely on the back end to resend messages and manage errors. Applications that send many repetitive messages, such as those reporting user position or other status, can often ignore transmission failures, because a subsequent message will be sent soon after the failure anyway.

- *Notify the user and let the user deal with the error*: This approach is appropriate for simple applications but can quickly frustrate users, especially if the application loses content you enter when the error occurs. It's far better to fail silently and notify the user of an abnormal condition passively (such as through a signal status icon) than it is to create a modal interface that forces the user to respond to various errors as they occur.

- *Queue messages for later delivery*: Applications requiring reliable message delivery should provide their own message queue, likely as records in a record store. This can gracefully handle most errors that can occur without user data loss.

Sending and Receiving MMS Messages

The exchange of MMS messages via the WMA is similar in principle to the exchange of SMS messages, although two wrinkles arise. First, as you saw in the "Managing the Multiple Parts of a MultipartMessage" section, sending or receiving an MMS message requires that you work with not only the message, but also the content that makes up the message. Second, the MMS protocol itself is a much more expensive protocol than the SMS protocol; sending or receiving an MMS message can take far longer than sending or receiving an SMS message, because MMS messages are larger and use the network's data channels more heavily than SMS does. This means that when using MMS, your application *must* use multiple threads to avoid stalling the user-interface thread. Moreover, your application should be prepared to receive more than one message at once, because heavily used applications may well be receiving a single MMS when another MMS message arrives.

Figure 14-5 shows the MMSMIDlet application, again running in two different instances. Unlike the SMSMIDlet example, the MMSMIDlet example sends an image rather than text.

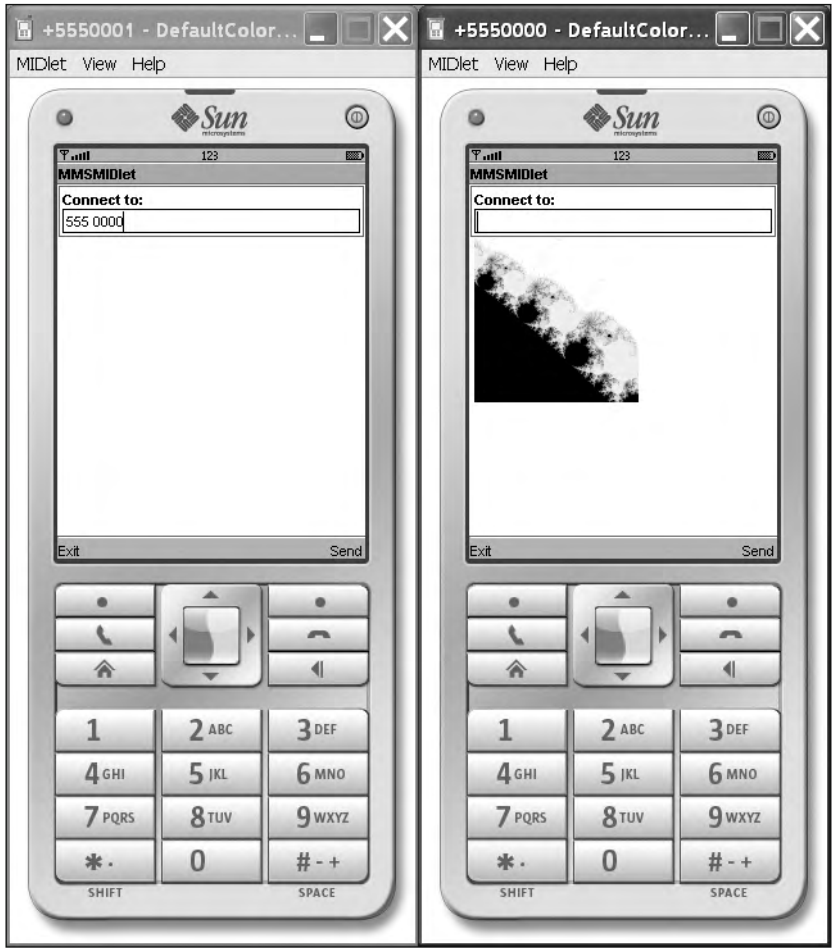


Figure 14-5. *Two MMSMIDlet instances running side by side in emulation*

Listing 14-8 shows the `MMSMIDlet` class, which comprises the user interface and receives functionality for the MMSMIDlet sample.

Tip As with Listing 14-6, Listing 14-8 doesn't compile independently. You also need the `MMSSender` class, which I show you in Listing 14-9.

Listing 14-8. *The MMSMidlet Class*

```
package com.apress.rischpater.mmsmidlet;

import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.wireless.messaging.*;
import java.io.IOException;

public class MMSMIDlet
    extends MIDlet
    implements CommandListener, Runnable, MessageListener {
    private MMSSender sender = null;
    private Command exitCommand = new Command("Exit", Command.EXIT, 2);
    private Command sendCommand = new Command("Send", Command.SCREEN, 1);

    private TextField numberEntry = null;
    private static final String FRACTAL_PATH = "/icons/fractal.png";
    private ImageItem imageItem = null;
    private Form form = null;
    private boolean endNow = false;
    private MessageConnection c = null;
    private final String appId = "MMSMIDlet";
    protected int msgAvail = 0;
    private final Integer monitor = new Integer(0);

    public MMSMIDlet() {
        sender = MMSSender.getInstance();
    }

    public void commandAction(javax.microedition.lcdui.Command c,
                              javax.microedition.lcdui.Displayable d) {
        if (c == exitCommand) {
            if (!sender.isSending()) {
                destroyApp(true);
                notifyDestroyed();
            }
```

```
    } else if (c == sendCommand) {
        String dest = numberEntry.getString();
            if (dest.length() > 0)
                sender.sendMsg(dest, appId, "A fractal!", FRACTAL_PATH);
    }
}

protected void destroyApp(boolean param) {
    try {
        endNow = true;
        c.close();
    } catch (IOException ex) {}
}

protected void pauseApp() {
    endNow = true;
    try {
        c.setMessageListener(null);
        c.close();
    } catch (IOException ex) {}
}

protected void startApp() {
    if (form == null) {
        form = new Form("MMSMIDlet");
        numberEntry = new TextField("Connect to:",
                                    null, 256,
                                    TextField.PHONENUMBER);
        imageItem = new ImageItem(null,null,
                ImageItem.LAYOUT_DEFAULT,null);

        form.append(numberEntry);
        form.append(imageItem);

        form.addCommand(exitCommand);
        form.addCommand(sendCommand);
        form.setCommandListener(this);
    }
    Display.getDisplay(this).setCurrent(form);
    startReceive();
}
```

```
    private void startReceive() {
        Thread t;
        try {
            c = (MessageConnection) Connector.open("mms://:" + appId);
            c.setMessageListener(this);
        } catch (Exception e) {}
        if (c != null) {
            t = new Thread(this);
            t.start();
        }
    }

    public void run() {
        Message msg = null;
        endNow = false;
        msgAvail = 0;

        while (!endNow) {
            synchronized(monitor) {
                if (msgAvail <= 0)
                    try {
                        monitor.wait();
                    } catch (InterruptedException e) {}
                msgAvail--;
            }
            try {
                msg = c.receive();
                if (msg instanceof MultipartMessage) {
                    MultipartMessage mpm = (MultipartMessage)msg;
                    MessagePart[] parts = mpm.getMessageParts();
                    if (parts != null) {
                        for (int i = 0; i < parts.length; i++) {
                            MessagePart mp = parts[i];
                            String type = mp.getMIMEType();
                            byte[] ba = mp.getContent();
                            if (type.equals("image/png")) {
                                Image image =
                                    Image.createImage(ba, 0, ba.length);
                                Display.getDisplay(this).callSerially(
                                    new SetImage(image));
                            }
                        }
                    }
                }
```

```
            } catch (Exception e) {}
        }
    }

    private void getMessage() {
        synchronized(monitor) {
            msgAvail++;
            monitor.notify();
        }
    }

    public void notifyIncomingMessage(MessageConnection c) {
        if (c != null) {
            getMessage();
        }
    }

   class SetImage implements Runnable {
       private Image img = null;
       public SetImage(Image i) {
           img = i;
       }
       public void run() {
           imageItem.setImage(img);
       }
   }
}
```

The division of responsibility in the `MMSMIDlet` class is the same as in the `SMSMIDlet` class, although both the receiving and sending code is more complicated, reflecting the asynchronous nature of MMS interaction. Initialization for message receipt in `startReceive` is similar to the implementation in `SMSMIDlet`, with one exception: the `MMSMIDlet` implementation registers a message listener with the `MessageConnection` returned from the GCF, so that the Java ME runtime can notify it of incoming messages through the `notifyIncomingMessage` method.

When the MIDlet receives notification of an incoming message, it invokes `getMessage`. Together, `getMessage` and the receive thread's `run` method (originally spawned by `startReceive`) use a message counter `msgAvail` and the monitor variable `monitor` to coordinate message receipt; if the MIDlet is receiving a message, `getMessage` waits before notifying the receive thread that another incoming message is pending.

The receive thread's main loop is slightly more complicated because it uses the monitor; it first waits on the monitor and then receives the incoming message. Receiving the message is the same as in the SMSMIDlet example—it just invokes `receive` on

the `MessageConnection` instance configured for MT messages—but message handling is trickier, because the message may contain any number of message parts. The receive thread checks to be sure that the `Message` it received from the `MessageConnection` instance is a `MultipartMessage` instance, and then examines each of its parts to find a part with the MIME type `image/png`. Once it does so, it creates an instance of `Image` using the bytes from that part. It then updates the display with the image from that part with the inner class `SetImage`. Finally, the thread's `run` loop resumes waiting on the monitor variable for another incoming message notification.

Like the SMSMIDlet example, sending also occurs on a separate thread; the `MMSSender` class manages the sending of each outgoing message. Listing 14-9 shows the `MMSSender` class.

Listing 14-9. *The MMSSender Class*

```
package com.apress.rischpater.mmsmidlet;

import javax.microedition.io.*;
import javax.wireless.messaging.*;
import java.io.*;
public class MMSSender implements Runnable {
    private static MMSSender me = new MMSSender();
    private MMSSender() {
    }

    public static MMSSender getInstance() {
        return me;
    }

    private String receiver = null;
    private String appId = null;
    private String image = null;
    private String msg = null;
    private String encoding = null;
    private boolean sending = false;

    public void sendMsg(String r, String id, String m, String i) {
        if (sending) return;
        receiver = r;
        appId = id;
        msg = m;
        image = i;
        encoding = System.getProperty("microedition.encoding");
```

```
    Thread t = new Thread(this);
    t.start();
}

public boolean isSending() {
    return sending;
}

public void run() {
    sending = true;
    try {
        sendMMS();
    } catch (Exception e) {}
    sending = false;
}

private void sendMMS() {
    String address = "mms://" + receiver + ":" + appId;
    System.out.println(address);
    MessageConnection c = null;
    try {
        c = (MessageConnection) Connector.open(address);
        MultipartMessage mpm = (MultipartMessage) c.newMessage(
            MessageConnection.MULTIPART_MESSAGE);
        mpm.setSubject("MMSMIDlet Image");
        if (image!=null) {
            InputStream is = getClass().getResourceAsStream(image);
            byte[] bImage = new byte[is.available()];
            is.read(bImage);
            mpm.addMessagePart(
                new MessagePart(bImage, 0, bImage.length,
                    "image/png", "id1", null, null));
        }
        if (msg!=null) {
            byte[] bMsg = msg.getBytes();
            mpm.addMessagePart(
                new MessagePart(bMsg, 0, bMsg.length,
                    "text/plain", "txt1", null, encoding));
        }
        c.send(mpm);
    } catch (Exception e) {}
```

```
        finally {
            if (c != null) {
                try {
                    c.close();
                } catch (IOException e) {}
            }
        }
    }
}
```

Identical in structure to the `SMSSender` class, `MMSSender` must do two things differently. First, it must determine the character encoding scheme used by the host Java ME virtual machine, so it can send it along with any outgoing MMS messages that include a text part. Second, it must manage multiple message parts. Although the user interface for MMSMIDlet only supports displaying the image part of an incoming MMS message, the `MMSSender` can send both a text part and an image part. By sending both a text part and an image part, I show you how to send multiple parts in a single `MultipartMessage` instance and thoroughly test the implementation of the MIDlet's message receipt functionality.

Getting the encoding scheme that the host virtual machine uses is easy; the `sendMsg` method just invokes the `System.getProperty` method with the value `microedition.encoding`. This returns the standard name of the encoding scheme, which you can then use when sending text parts of MMS messages. Once the `sendMsg` method captures its incoming arguments, it creates and starts a new thread to send the outgoing message contents.

The thread uses the private `sendMMS` method to actually perform the sending; this method must create `MessagePart` instances for each of the indicated payloads, add those parts to a new `MultipartMessage` instance, and send the `MultipartMessage`. For each of the text and image parts, the method first obtains an array of bytes representing the payload. For an image payload, it fetches the bytes from the resource in the MIDlet JAR file; for a string payload, it fetches the bytes from the `String` containing the message. Then, it creates a new `MessagePart` instance, specifying the MIME type and a name for the part when it creates the part. When creating the `MessagePart`, the `sendMMS` method doesn't specify a location for the part, because there is no location. In addition, the `MessagePart` instance that contains the text of the message to send also includes the encoding originally fetched from the Java ME runtime. Once `sendMMS` creates the `MessagePart` instance with the desired data, it adds the message part to the outgoing message. With both message parts added to the outgoing message, the code uses the `MessageConnection`'s `send` method to send the message, just as it would any other WMA message.

■ **Note** This example sends an image included with the MIDlet at compile time. Generating and encoding images dynamically to send with the WMA can be difficult. The Mobile Media API (MMAPI), which I discuss in Chapter 16, can generate JPEG images from a device's camera, but you generally cannot encode `Image` objects to send with the WMA over MMS without additional code such as a PNG encoder.

Wrapping Up

The WMA, defined first in JSR 120 and extended in JSR 205, defines an extension to the GCF that lets MIDlets send and receive SMS and MMS messages, as well as receive SMS-CB messages. In essence, using it is the same as writing any other GCF-aware application: you use the `Connector`'s `open` method as a factory for `MessageConnection` objects, which you can then use to create instances of specific `Message` subclasses to represent messages for the device to send or messages that the device has received.

The `MessageConnection` instance you create with the GCF acts as a factory for messages; you can create new MO messages using its `newMessage` method, passing the kind of message you want to create, or you can wait for an incoming message by invoking its `receive` method (which blocks the current thread until a message arrives). The WMA defines the `Message` class to encapsulate the notion of an address and timestamp for messages. Subclasses of `Message` define specific semantics for text SMS messages, binary SMS messages, and MMS messages: the `TextMessage`, `BinaryMessage`, and `MultipartMessage` classes, respectively. Using these classes, you can set the message payload, or in the case of `MultipartMessage` instances, you can manage multiple recipient addresses as well as the various parts that constitute a message. Once you configure an outgoing message, you can send it using the `MessageConnection`'s `send` method.

Using the WMA requires that your MIDlet have privilege. When packaging your application, be sure to include the privileges for both the kind of message your application uses as well as the direction (send or receive) of the messages your MIDlet uses in its JAD file.

Intermezzo

As you've read this book, you've progressed from the basics provided by all implementations of Java ME through the configurations, profiles, and packages that constitute today's commercially successful Java ME marketplace. You may have skipped Part 2 or Part 3, depending on your interest in a specific configuration of Java ME.

In the pages that follow, I discuss some of the most important extensions to the Java ME platform. Some, such as the MMAPI, can be found on nearly all the Java ME devices in production today. Others aren't so prevalent, either because they're new additions to the Java ME landscape or because manufacturers have had less impetus to bring these APIs to developers and consumers. Unlike previous parts, which I intended for you to consume and digest as a main course, these chapters are organized differently. Each is intended to stand on its own and give you a sampling of the various ways the Java ME platform grows to meet needs. Enjoy, and bon appétit!

PART 5

Other Java ME Interfaces

In the last four parts, you've learned the fundamentals of building applications using the two Java configurations supporting Java ME: the CLDC and the CDC. Along the way, you've seen most of the interfaces and libraries common to the Java ME platform that make it so powerful: the Java language and basic Java interfaces; GUI support through the MIDP, the AWT, and the AGUI; support for files through Java's existing mechanism or the FCOP; communication via HTTP and wireless protocols; and other features. Much of what you've seen belongs squarely to the Java ME platform; some bits, such as the FCOP and PIM APIs and support for web services and wireless messaging, are actually extensions to the Java ME platform so common as to be encountered on nearly all Java ME devices. Others, like the advanced graphics toolkits provided by the Personal Profile and the AGUI, are less common, but it's increasingly likely that you'll encounter them in the future.

However, the rest of this book isn't about Java ME itself, but rather about some of the most powerful and exciting packages that sit atop Java ME. As I write this, there are more than 20 JSRs that define optional packages that enhance the Java ME platform. They provide interfaces that support additional capabilities such as interfaces for cryptography, support for video, and location-based services. Not all of the packages defined by the JSR process are available on all Java ME devices, but the growing convergence between all devices makes it likely that given your application idea, you'll find a JSR documenting a Java package for Java ME that implements the interfaces you need, and there will be sufficient devices with support for that package. As a result, you'll likely have a viable market for your application.

CHAPTER 15

Securing Java ME Applications

Security plays a big part in the success of today's mobile marketplace. More so than ever before, consumer devices are being woven into the very infrastructure of electronic commerce. This revolution is being powered by mobile versions of the same security and trust technologies—largely powered by the revolution in cryptography over the last generation—that power electronic commerce on the Internet.

In this chapter, I look at the some of the various components available to Java ME developers that can help you create more secure applications. After beginning with a look at why you should have a grasp of today's security fundamentals, including a review of some key building blocks when designing secure applications, I move to a discussion of Java ME's Security and Trust Services API—an optional package that provides support for smart cards and cryptography. I next touch on Java ME's relatively new Contactless Communication API, which enables secure commerce applications through near-field communication devices to enable using your mobile device as a wireless wallet. Finally, I close with a discussion of the Bouncy Castle cryptography package—a full-featured, open source package for providing cryptography for Java ME devices.

Understanding the Need for Security

Back when I wrote my first networked application, the Internet was a small place—not so small that everybody knew everybody, but still small enough that passwords were often transmitted in the clear, right there where everybody could read them, if that's what they intended. Of course, that was before the Web, too, and writing a networked application required serious protocol work or a good understanding of the remote procedure call semantics just then in vogue for networked computing. Fast-forward a couple of decades, and the Internet's no longer a wild and wooly frontier, but a teeming metropolis. While it's tempting to blame the influx of people for the need for greater security, the truth is far more complex and as much a reflection of human nature as one of human presence. Today, securing an application is often as much an important part of gaining

the trust of the stakeholders involved as it is in repelling (real or imagined) attacks on one's intellectual property and commercial resources, such as content, storage, services, and personal information.

Designing a secure application requires both an in-depth understanding of the risks to your application as well as the countermeasures you can adopt to mitigate those risks. Put simply, a *risk* is any possible event that can cause a loss. Risks are associated with *threat*—a method of triggering a risk event. The following are examples of threats of risk:

- A user gaining access to your application without paying for it
- A user's personal data (such as identity information) being given to an unauthorized third party, with or without your knowledge
- A third party masquerading as you when interacting with your customers

In today's highly networked world, most people pay attention to the risks that are made manifest by networked applications, but of course that need not be the only ones that apply. Consider the case of a user losing her mobile device and a third party accessing its record store to obtain personal data.

You address threats through the adoption of *countermeasures* that attempt to stop a threat from triggering a risk, such as the following:

- Only offering your application and content through a trusted distribution service to devices that prevent redistribution
- Encrypting vulnerable personal data before transmitting it across the network or storing it on the device's record store
- Using secure network protocols such as HTTPS with signed certificates to verify identity when interacting with customers

Central to the design of countermeasures is the notion of cost: when you select countermeasures, you aim to make it more expensive for a hypothetical opponent to perform some task than the task is worth after factoring in the likelihood that the actual risk event will take place. You must strike a fine balance between the costs that security imposes (higher complexity, larger footprint, and greater user complexity) and the cost of a realized security threat.

A key tool in today's efforts to secure applications is *cryptography*: the art and practice of hiding information. Cryptographic solutions, largely based on recent advances in number theory, offer several tools for addressing security threats, including the following:

- *Ciphers*: A cipher lets you encrypt data or decrypt data once it's been encrypted. An encrypting cipher takes your original message, called *plaintext*, and renders it unreadable to third parties without the corresponding decrypting cipher. Ciphers rely on *keys*; to decipher the encrypted message, you must have a key. Today's ciphers use advancements in number theory that let you split keys into a public part and a private part; participants sharing a secret message keep their private key to themselves and use a recipient's public key to encrypt a message.

- *Message digests*: A message digest is nothing more than a cleverly designed hash function; it takes a large block of data and reduces it to a small block of data. Two popular message digest functions are SHA-1, which creates a digest 20 bytes long for an arbitrary input, and MD5, which creates a digest 16 bytes long for an arbitrary input.

- *Digital signatures*: A digital signature is a personalized message digest. It uses an individual's private key to create a message digest that can be verified using the signer's public key, to prove that the person generating the signature signed the message.

- *Certificates*: A certificate is really just an extension of a digital signature—it's a document signed by a trusted third party that proves your identity.

■ **Note** For comprehensive coverage of cryptographic concepts and algorithms, you can't do better than *Applied Cryptography: Protocols, Algorithms, and Source Code in C* by Bruce Schneier (John Wiley & Sons, 1995).

These tools are in widespread use in the mobile marketplace today. Consider digital signatures and certificates: they provide the backbone for today's on-deck delivery of wireless applications. Through the Java Verified Program, a third party signs your application before giving it to carriers for distribution; when a device downloads your application, it verifies the signature, permitting access to restricted APIs based on the authorities that signed your application.

Another application of these tools is HTTPS, based on TLS. HTTPS is the secure version of HTTP that you may well be using to secure your application's network. HTTPS can rely on certificates exchanged between client and server to prove the identify of both; regardless, it uses public-key ciphers to encrypt the network communication that occurs between client and server.

Of course, cryptography is but one countermeasure among many. Designing a secure application requires *defense in depth*—never relying on a single security measure alone. Strategies including physical security (preventing access to your application service's

data center, for example), logging, auditing, and using secure network channels help you secure your networked application. Bear in mind that cryptography can solve only one segment of the security challenges you face, and that many of the challenges you face may be social (think of phishing scams prevalent in today's e-mail) rather than technical.

■ **Note** *Secrets & Lies: Digital Security in a Networked World* by Bruce Schneier (John Wiley & Sons, 2000) and *The Art of Deception* by Kevin D. Mitnick and William L. Simon (Wiley Publishing, 2002) are good places to start learning more about the technical and social threats to application security and how to establish countermeasures against those threats.

Looking at Java ME's Security and Trust Services

Java SE provides a host of cryptographic interfaces including Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE); not so with the base implementation of Java ME. There are a number of reasons for this, both technical and legal; memory footprint and computational complexity make cryptography difficult for some Java ME devices, and export restrictions could hobble Java ME's adoption in some markets if it came bundled with strong cryptographic solutions.

The extensible nature of Java ME comes to the rescue, however. JSR 177 defines the Security and Trust Services API for J2ME (SATSA). In addition to supporting cryptographic operations, SATSA includes optional components for Java ME that provide APIs for communication with hardware security components, as well as certificate and signature management. SATSA defines four key optional packages that a device may support:

- *Application Protocol Data Unit (APDU)*: This communications API enables low-level communication between your application and cryptographic hardware such as Java smart cards.
- *SATSA-Java Card RMI (SATSA-JCRMI)*: This communications API permits high-level communication between your application and a Java smart card.
- *SATSA-public key infrastructure (SATSA-PKI)*: This API provides support for managing public keys.
- *SATSA-CRYPTO*: This API provides a subset of the `java.security` package for cryptography.

The APDU protocol builds on the GCF to provide a general interface for communicating with Java smart cards, Universal Subscriber Identity Modules (USIMs), and

security token smart cards. Suitable for low-level communications, the APDU interface deals at the level of individual bytes exchanged with cryptographic hardware; for many applications, the SATSA-JCRMI API may be more appropriate. The other two APIs—SATSA-PKI API and SATSA-CRYPTO—provide high-level cryptographic services to your application.

■ **Caution** As I write this, there is no unified SDK that implements all of these optional packages for either NetBeans or the older Sun Java Wireless Toolkit. Worse, not only are the packages described in JSR 177 optional, but a vendor may choose to only implement *some* of the packages described in JSR 177. This has the potential for serious fragmentation of the availability of these APIs.

Communicating with Cryptographic Hardware Using the APDU API

Today's smart cards secure many popular consumer electronics devices, including some phones, computers, and consumer electronics set-top boxes. They're also a key security feature of some industrial communications control systems. At its heart, a smart card is simply a small plastic card with one or more hardened embedded circuits inside, providing memory and possibly processing power. Smart cards typically include tamper-resistant properties such as a secure file system, a secure microprocessor, and human-visible tamper-indicating devices such as holograms or other information. Some smart cards include support for cryptographic functions such as key generation and cipher algorithms; the purpose of the SATSA API is to support accessing these cards using Java ME–enabled hardware.

At the lowest software level, hardware such as a Java smart card or USIM communicates with its host hardware using a channel of bits and bytes, similar to a network socket or serial port. The International Organization for Standardization (ISO) provides ISO 7816 to describe the mechanical, electronic, and software communication scheme between smart cards and their host hardware. The software scheme relies on the notion of application identifiers (AIDs) that permit the selection of a specific smart card function; communication to the smart card is accomplished using an ISO-defined protocol of application protocol data units (hence the name *APDU* for the protocol used by the SATSA package to communicate with the card).

Because this low-level access to card functions consists of the exchange of bytes across a communications channel, you might think the Java GCF would provide an excellent means to support this communication, and you'd be correct. Figure 15-1 shows the relationship between the GCF classes and the APDU hierarchy.

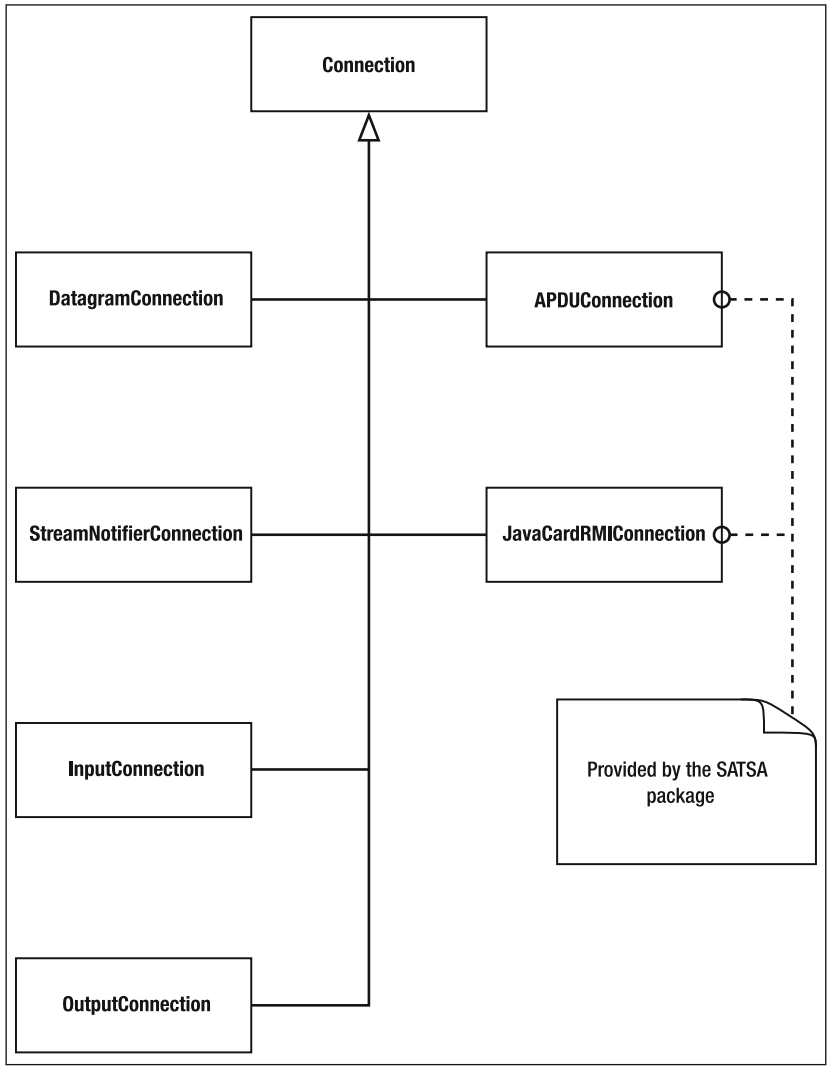


Figure 15-1. *The APDU classes in relationship to the Java ME GCF hierarchy*

Two interfaces—`javax.microedition.apdu.APDUConnection` and `javax.microedition.jcrmi.JavaCardRMIConnection`—provide the means by which your application can communicate with a smart card. As its name suggests, you use the `APDUConnection` to perform low-level APDU interaction with the card, while you use the `JavaCardRMIConnection` to perform Java RMI to Java smart cards—a topic I touch on in the next section, titled "Communicating with Java Smart Cards Using JCRMI."

Just as when using the GCF to obtain access to FCOP or network interfaces, you begin with a Uniform Resource Identifier (URI) that describes the entity to which your application will connect. For smart cards, the URI has this format:

```
protocol:[slotid]; target
```

Each part is defined as follows:

- `protocol`: Either `apdu` for an APDU-based connection (returning an object implementing `APDUConnection`) or `jcrmi` for a JCRMI-based connection (returning an object implementing `JavaCardRMIConnection`)
- `slotid`: An integer indicating the slot into which the user has inserted the card
- `target`: The application identifier for the specific smart card application with which to connect, or the word `SAT` to connect to a SIM application toolkit

Because devices can support more than one card slot, you can query smart card–enabled hardware using the `System.getProperty` method with the `microedition.smartcardslots` property, which returns a comma-delimited list of identifiers for existing card slots. Each identifier consists of two characters: the first indicates the index you use when specifying a URI to indicate the slot, and the second is either a `C` or an `H` indicating whether the slot is cold-swappable or hot-swappable—that is, whether cards can only be removed when the device is off (cold-swappable) or while the device is in operation (hot-swappable).

Once you establish a connection to the smart card, you perform your I/O using the `exchangeAPDU` method. Using `exchangeAPDU`, you send command APDUs to the smart card and receive the response as an array of bytes. If necessary, you can establish several logical channels to the smart card using multiple `APDUConnection` instances, letting you use more than one application on the smart card at a time. For example, you might write the code shown in Listing 15-1.

Listing 15-1. *Exchanging an APDU with a Smart Card*

```
APDUConnection ac;
try {
    byte[] commandAPDU = …;
    String url = "apdu:0;AID=A1.0.0.67.4.7.1F.3.2C.5";
    ac = (APDUConnection)Connector.open(url);

    // Send a command APDU and receive a response APDU
    byte[] responseAPDU = ac.exchangeAPDU(commandAPDU);

    // Process the response
    …
} catch(Exception e){
    // Handle Exception
    …
```

```
} finally {
   …
   if (ac != null) {
      // Close connection
      ac.close();
   }
}
```

Of course, you should be prepared to handle exceptions; interruptions can occur, such as a user ejecting a smart card while in use. When you're finished, you must also close the connection to release the underlying resources used by the Java VM to connect to the smart card.

The `APDUConnection` interface also provides support for enabling, disabling, changing, and confirming the personal identification number (PIN) that the smart card's owner uses to authenticate himself before permitting the smart card to perform an operation. The following methods are used to interact with the user and return a smart card response after processing a PIN:

- `changePin`: Prompts the user to enter the current PIN for the card and then a new PIN for the card, and commits the PIN change to the card if appropriate
- `disablePIN`: Prompts the user to enter the current PIN for the card and disables the need for subsequent identification using a PIN
- `enablePIN`: Prompts the user to enter the current PIN for the card and enables requiring a PIN for subsequent use of the card
- `enterPIN`: Prompts the user to enter the current PIN for the card and verifies the PIN

To open an `APDUConnection`, your application must run with privilege; the privilege you must specify is `javax.microedition.apdu.aid` to an arbitrary smart card application ID, and `javax.microedition.apdu.sat` to a USIM toolkit. Typically, this permission is only granted to applications in the operator, manufacturer, or third-party trusted domains.

Communicating with Java Smart Cards Using JCRMI

The `APDUConnection` provides the lowest level of access to a smart card. This level of access isn't for the faint of heart; it requires a firm understanding of both smart card fundamentals and the application communication protocol for a specific application on a smart card. Such an interface is ripe for abstraction, and that's the purpose of JCRMI.

JCRMI uses concepts from Java's RMI architecture to permit access to smart card operations on Java-enabled smart cards directly using Java (see Chapter 11 for a deeper

understanding of how RMI works in Java). Instead of working at the level of individual bits and bytes, you can connect to a Java smart card application on the card, receive a stub interface to the application, and invoke its methods remotely, as shown in Listing 15-2.

Listing 15-2. *Using JCRMI to Connect with a Java Smart Card*

```
try {
    String url = "jcrmi:0;AID=A0.0.0.67.4.7.1F.3.2C.3";
    JavaCardRMIConnection jc =
        (JavaCardRMIConnection)Connector.open(url);
    Wallet wallet = (Wallet)jc.getInitialReference();
    …
    currentBalance = wallet.getBalance();
    …
    jc.close();
 } catch (Exception e) {
    …
 }
```

In essence, the combination of the GCF `Connector` class and the AID you specify acts as the registry for a remote Java object; you use the `JavaCardRMIConnection` instance's `getInitialReference` method to obtain a reference to the stub representing the remote application on the card.

As with an `APDUConnection`, exceptions can occur; this can happen if the card is not inserted, if the smart card application does not exist, or if the Java ME application is not permitted to use the smart card interface at all. Note that if the card is ejected while the remote object is in use, access to the remote object of course fails; reinserting the card requires the Java ME application to reconnect to the smart card application.

The JCRMI interface is significantly more limited in the following ways than either Java RMI or Java RMI OP:

- Remote classes can only implement a maximum of 15 interfaces.
- Only primitive Java types (`boolean`, `byte`, `short`, `int`, and single-dimensional arrays of these types) can be returned by remote methods.
- Parameters and return values are exchanged by value, except for remote object references.

As with the `APDUConnection`, using a `JavaCardRMIConnection` instance requires permission; you'll want to assert the `javax.microedition.jcrmi` permission in your application's JAD file or manifest. As with the `APDUConnection`, use of this privilege is restricted to applications in the operator, manufacturer, or third-party trusted domains.

Leveraging the SATSA High-Level APIs for Cryptography

While the two optional packages defined by SATSA that you've seen are for interfacing with cryptographic hardware, the remaining two provide implementations of common cryptographic operations needed by many of today's mobile applications. The SATSA-CRYPTO package includes a subset of the `java.securtity` package, a subset of the `java.security.spec` package, a subset of the `javax.crypto` package, and a subset of the `javax.crypto.spec` package to provide APIs for public and private key management, message digests, signature verification, and data encryption. The SATSA-PKI security packages include `javax.microedition.pki` and `javax.microedition.securityservice`, which define classes to support basic user-certificate management.

Using the SATSA-CRYPTO API, let's look at two common operations you're likely to perform at some point during application development: creating message digests and encrypting (or decrypting) a message.

Using the SATSA-CRYPTO API to Create a Message Digest

One of the most common cryptographic operations a mobile application may be required to perform is creating a message digest. Many web service APIs today use message digests as a means to prevent tampering with the payload of a web service request or response; the message digest may appear as a separate HTTP header or an argument to the web service request, or simply may be appended to the web service header. This couldn't be easier than using the SATSA-CRYPTO API, as shown in Listing 15-3.

Listing 15-3. *Creating a Message Digest*

```
String webRequest = "…";
byte[] message = webRequest.getBytes();
static String digestAlgorithm = "MD5";
static int digestLen = 16;
byte[] digest = new byte[digestLen];

try {
    java.security.MessageDigest md;
    md = java.security.MessageDigest.getInstance(digestAlgorithm);
    md.update(message, 0, message.length);
    md.digest(digest, 0, digestLen);
} catch (Exception e) {
    // Handle NoSuchAlgorithmException or DigestException
    …
}
```

The SATSA-CRYPTO API provides the `MessageDigest` class, which provides both a static factory for creating message-digest algorithm implementations as well as an interface to the message-digest algorithm. You can obtain a concrete instance of `MessageDigest` by invoking its `getInstance` function and passing the name of the digest function (one of `"MD5"` or `"SHA-1"`) whose implementation you want. The resulting `MessageDigest` object consumes arrays of bytes you pass to it by calling its `update` method, providing the resulting message digest and resetting the algorithm when you invoke its `digest` method.

This code opens assuming that you have a web request in hand for which you'd like to generate an MD5 digest in `webRequest`. Because the `MessageDigest` interface works with arrays of bytes, the code first gets the representation of the web request as an array of bytes and then declares variables to contain the resulting digest. With this done, it gets an instance of the MD5 `MessageDigest` implementation and passes it the bytes that make up the `webRequest` by invoking `md.update`. For a long document, you could invoke this multiple times. Once the `MessageDigest` instance has been fed the message using `update`, the code asks it to compute the digest using the `digest` method, which takes an array of bytes to store the output and the number of bytes in the array.

The `MessageDigest` implementation that SATSA-CRYPTO provides typically includes both the MD5 and SHA-1 algorithms, but there's no guarantee of either being available. Consequently, your application should be prepared to handle the `NoSuchAlgorithmException`.

You can also use the SATSA-CRYPTO API to verify message digests by creating a message digest of a source document and comparing it with a provided message digest. There's no API for comparing message digests; simply loop over the bytes in each digest and compare individual bytes to determine if the message digests are equivalent.

Encrypting and Decrypting Using the SATSA-CRYPTO API

Under most circumstances, the only time to encrypt data is when it leaves a device for transmission on the network. In this case, HTTPS is generally sufficient for your data security needs; occasionally you may find a need to store data locally in an encrypted form or exchange data using a protocol other than HTTPS. The SATSA-CRYPTO API provides interfaces for encrypting and decrypting data, with one limitation: while you can use public-key (also called *asymmetric*) cryptography to encrypt data using the SATSA-CRYPTO API, you cannot decrypt data encrypted using public-key cryptography.

It's best to think of the SATSA-CRYPTO API for ciphers as an interface and not as a concrete implementation. For a given target, there's no clear guarantee of *which* cipher implementations you'll encounter; one device may support Data Encryption Standard (DES), Rivest Cipher 4 (RC4), and the Encryption Standard (AES), while another might only support DES. Regardless of what's supported, the general approach to using the SATSA-CRYPTO cipher interface remains the same:

1. Generate the plaintext you want to encrypt as an array of bytes.
2. Generate the key you want to use to encrypt your plaintext, again as an array of bytes.
3. Get an instance of the desired cipher implementation, which will be an instance of `java.security.Cipher`.
4. Initialize the cipher.
5. Either encrypt the plaintext progressively using the `Cipher`'s `update` method, or perform the encryption in a single operation using the `Cipher`'s `doFinal` method.

Predictably, decryption is the reverse of encryption; instead of specifying a cipher in step 3, specify the cipher and indicate that it should be used in decryption mode. Instead of passing plaintext in step 5, pass the enciphered text. Consider Listing 15-4, which demonstrates encrypting a message.

Listing 15-4. *Encrypting a Message*

```
String algo= "RC4";
byte[] secretKey =   { … };
String plainText = "Here there be treasure";
byte[] plainTextBytes = plainText.getBytes();

try {
    java.security.Key key =
        new SecretKeySpec(secretKey, 0, secretKey.length, algo);
    java.security.Cipher cipher;

    cipher = Cipher.getInstance(algo);
    cipher.init(Cipher. ENCRYPT_MODE, key);

    int ciphertextLength = plainText.length();
    byte[] cipherTextBytes = new byte[ciphertextLength];

    cipher.doFinal(plainTextBytes, 0, plainText.length, cipherTextBytes, 0);
} catch (Exception e) {…}
```

This follows the algorithm I outlined step for step; note especially the use of `ENCRYPT_MODE` as the selected mode when initializing the `Cipher` instance. Decryption would be the reverse of encryption, and the code would just need to pass `DECRYPT_MODE` for the corresponding argument to `init`. After this code has run, assuming there are no exceptions, the byte array `cipherTextBytes` will contain the encrypted text.

Ciphers can differ in how they handle their input; some, like DES, are *block ciphers*, which require the input to come in regularly sized blocks, such as blocks of 64 bytes. When using block ciphers, you must pad the input to block aligned boundaries; complicating this, some block ciphers are asymmetric, meaning that the input block size and output block size are different. Other ciphers, like RC4, are stream ciphers: they just take a stream of bytes (plaintext or encrypted text) and work their magic. When using a block cipher, be sure you pass whole blocks to the `update` and `doFinal` methods.

A number of things can happen when encrypting or decrypting using the `Cipher` interface; these are signaled using one of the following exceptions:

- The API throws `BadPaddingException` or `IllegalBlockSizeException` for block ciphers when the input is not padded correctly or occurs in the wrong-sized block.
- The API throws `InvalidKeyException` when an invalid key is used with a cipher.
- The API throws `IllegalStateException` when you attempt to use the cipher to encrypt or decrypt a message before initializing it.
- The API throws `NoSuchAlgorithmException` if you attempt to get an instance of `Cipher` for an algorithm that is not supported.

■ **Caution** In the real world, you shouldn't be so cavalier with a cipher's private key as to embed it in source code or another easily read resource such as a component of your JAR file. Malicious users could find the key by decompiling your application or JAR. Instead, you could create the secret key at distribution time using a different key for each application, and only distribute the application JAD using secure HTTP. However, this would mean that your application couldn't be signed and verified by participants of the Java Verified Program. You could also provide a web service that provides secure key generation and delivery over HTTPS to your application.

Exploring the Bouncy Castle Solution to Security Challenges

While optional solutions like SATSA can solve problems for many devices, relying on an optional solution doesn't fulfill the write-once, run-anywhere promise that Java holds. Fortunately, there's a solution that runs on Java ME: the Bouncy Castle Java cryptography API. Developed in Australia and available under a liberal open source license, the API provides clean-room implementations of a provider for JCE and JCA for Java SE,

and a lightweight cryptography API that fits well with Java ME. This API implements many popular ciphers, including DES, AES, Blowfish, International Data Encryption Algorithm (IDEA), and others, as well as a slew of message-digest algorithms, including MD5 and SHA. The API also provides a large number of utilities, including support for Abstract Syntax Notation One (ASN.1) encoding and decoding, X.509 cryptographic certificates and public-key exchange files, and codecs for Secure/MIME (S/MIME) and OpenPGP.

You can get a copy of the API from `http://www.bouncycastle.org/java.html`; as I write this, the current version is 1.39. You'll want to download the release, unpack it, and drop the ZIP files in the `zips` directory in the `lib` directory of your project.

The Bouncy Castle API consists of a number of packages; the following are the ones you're most likely to use:

- `org.bouncycastle.crypto`: Includes base classes that represent cryptographic engines, digests, and other basic constructs
- `org.bouncycastle.crypto.digests`: Includes classes that implement message-digest algorithms
- `org.bouncycastle.crypto.engines`: Includes classes that implement cipher algorithms
- `org.bouncycastle.crypto.generators`: Includes classes that implement key generators
- `org.bouncycastle.crypto.params`: Includes classes that implement representations to options for specific cipher algorithms

It's worth noting that the full implementation of Bouncy Castle for Java ME is *large*: nearly one and a half megabytes! Of course, almost no application needs all of the functionality provided by the API. To manage the size of applications depending on the API, you need to use an obfuscator—something I discuss in Chapter 3 in the section titled "Building CLDC/MIDP Applications." NetBeans includes the ProGuard (`http://proguard.sourceforge.net`) obfuscator, which does a fine job at managing the output application size. Figure 15-2 shows appropriate settings for use with the Bouncy Castle API; choose Obfuscating under the Build property.

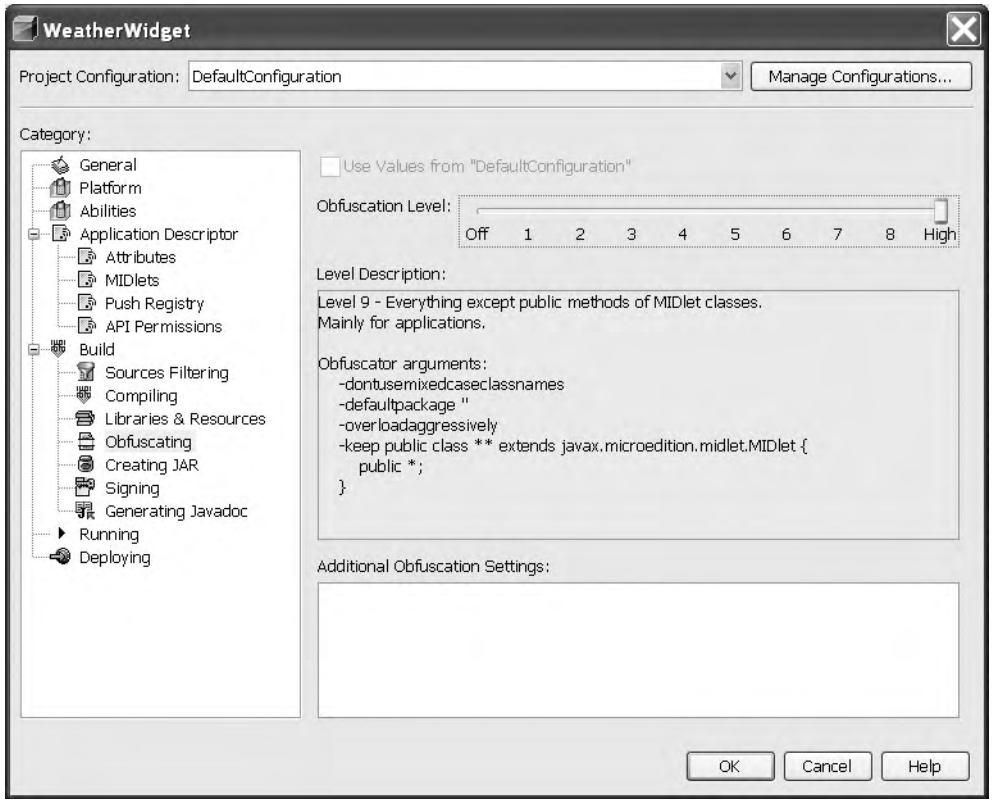


Figure 15-2. *Representative obfuscation settings for a Java ME project using the Bouncy Castle API*

The settings shown in Figure 15-2 instruct the obfuscator to use the following options:

- `-keep public class *.* extends javax.microedition.midlet.MIDlet`: Forces MIDlet subclasses to retain their names
- `-dontusemixedclassnames`: Doesn't allow mixed case in generating obfuscated class names, to work around an issue in Microsoft Windows
- `-defaultpackage ''`: Renames all packages other than packages implementing the `MIDlet` interface
- `-overloadaggressively`: Uses the same name for different methods whenever possible (that is, whenever different methods have different arguments and return values), creating a smaller package

■ **Note** When obfuscating packages that use the Bouncy Castle API, the `-defaultpackage ''` directive is especially important, because Bouncy Castle provides some classes in the `java` package, such as `java.math.BigInteger`. The class loader for the CLDC doesn't load classes defined in system packages, because doing so is a security threat (it would enable a rogue application to replace a system class with one of its own). By using the `-defaultpackage ''` directive, the obfuscator renames any classes in that package, so this isn't a problem.

Let's see how using the Bouncy Castle API compares with using the SATSA APIs to create message digests and encrypt a message.

Creating Message Digests Using the Bouncy Castle API

Message digest functions are provided by the `org.bouncycastle.crypto.digests` package, which implements various message-digest algorithms in a manner consistent with the JCA. Digest algorithms must implement the interface defined by `org.bouncycastle.crypto.Digest`; this resembles the JCA's `MessageDigest` interface also found in the SATSA-CRYPTO API. You might write the code in Listing 15-5 to compute an MD5 digest for a web request.

Listing 15-5. *Computing an MD5 Digest for a Web Request*

```
String webRequest = "…";
byte[] message = webRequest.getBytes();
byte[] digest;
org.bouncycastle.crypto.Digest md =
    new org.bouncycastle.crypto.digests.MD5Digest();
md.update(message, 0, message.length);
digest = new byte[md.getDigestSize()];
md.doFinal(digest, 0);
```

The logic of this code is very similar to the code you'd write using the JCA or SATSA-CRYPTO API. At a high level, the approach is the same: create an instance of the digest algorithm, use the `update` method to pass it the bytes from which to compute the digest, and then compute the digest using the `doFinal` method. However, there are a few differences:

- There's no generic factory for digest algorithms; instead, you explicitly create an instance of the digest algorithm you need (this helps limit the number of classes included in your application when you link against the Bouncy Castle implementation).

- The message-digest algorithms don't throw exceptions to signal errors.
- The `Digest` interface provides a method that tells you how many bytes long the digest will be.

In addition to the handy `getDigestSize` method provided by the `Digest` interface, you can also get a human-readable name of a message-digest algorithm by invoking the `Digest` method `getAlgorithmName`, which you could present to the user via your application's UI. Most of the message-digest algorithms the Bouncy Castle API provides actually implement the `ExtendedDigest` interface, which implements `Digest` and adds the `getByteLength` method. You can invoke `getByteLength` to learn the size of the internal buffer the digest applies its algorithm to.

Encrypting and Decrypting Using the Bouncy Castle API

The Bouncy Castle API provides cipher implementations through cryptographic engines (in `org.bouncycastle.crypto.engines`) that implement specific interfaces such as `AsymmetricBlockCipher`, `BlockCipher`, or `StreamCipher` (all of which you can find in the `org.bouncycastle.crypto` package). These interfaces all serve a common purpose: they let you initialize the cipher, provide data in the form of byte arrays to be encrypted or decrypted (as either blocks or part of a message stream), and then perform the encryption or decryption operation. As with the JCA and SATSA-CRYPTO API, when you initialize a cipher, you indicate whether you want the implementation to perform encryption or decryption, as well as the details of the key for the operation. Using the Bouncy Castle API to perform an encryption with RC4, you might write the code shown in Listing 15-6.

Listing 15-6. *Using the Bouncy Castle API to Perform RC4 Encryption*

```
byte[] secretKey =   { … };
String plainText = "Here there be treasure";
byte[] plainTextBytes = plainText.getBytes();
org.bouncycastle.crypto.StreamCipher cipher =
    new org.bouncycastle.crypto.engines.RC4Engine();
    cipher.init( true,
        new org.bouncycastle.crypto.params.KeyParameter(secretKey));
byte[] cipherTextBytes = new byte[plainTextBytes.length];

try {
    cipher.processBytes( plainTextBytes, 0,
                         plainTextBytes.length,
                         cipherTextBytes, 0 );
} catch( Exception e ) {…}
```

As with message digests, the interface to the Bouncy Castle API is conceptually similar to the JCA and the SATSA-CRYPTO API but not exactly the same. Again, instead of using a factory to create the desired cipher engine, you simply instantiate the engine directly. You initialize the resulting engine similarly, specifying whether you want the engine to encrypt (pass `true`) or decrypt (pass `false`) along with options for the engine to its `init` method. Finally, you pass the bytes to be encrypted or decrypted to the engine for processing. How the engine accepts the bytes you want it to process depends on whether the engine implements a block cipher or stream cipher. You pass bytes to a block cipher or asymmetric block cipher for processing using the `processBlock` method, passing a complete block each time. You can determine the block size of a block cipher by invoking its `getBlockSize` method, or an asymmetric block cipher by invoking either `getInputBlockSize` or `getOutputBlockSize` for the input or output block size, respectively. You pass bytes to a stream cipher using the `processBytes` method.

In either case, the cipher engine returns the result to the method you invoked for processing. This method may throw one of a number of exceptions, including the `DataLengthException` if the data array is of an invalid length, or the `IllegalStateException` if you failed to initialize the cipher.

Although this example uses the (relatively weak) RC4 algorithm, a number of other supported ciphers are packaged with the Bouncy Castle API. Unlike the SATSA-CRYPTO API, the algorithms are implemented for both symmetric and asymmetric ciphers, so applications requiring both encryption and decryption of public-key ciphers benefit from using the Bouncy Castle API instead of the SATSA-CRYPTO API.

In addition to supporting many different ciphers, the Bouncy Castle API also provides key-generation algorithms for the ciphers it supports. This is an important feature of the API, because secure key generation and distribution is a challenge when writing applications. The classes contained by `org.bouncycastle.crypto.generators` include generators for supported ciphers; creating a key is as simple as creating the appropriate generator and invoking a method. For example, to create a random key for DES, you might write the code shown in Listing 15-7.

Listing 15-7. *Generating a Random Key*

```
org.bouncycastle.crypto.generators.DESKeyGenerator generator =
    new org.bouncycastle.crypto.generators.DESKeyGenerator();
byte[] key = generator.generateKey();
```

Some key generators must be initialized first, so be sure to check the documentation for the cipher system and key generator you choose. These key generators are usually those that provide keys for public-key cryptography, in which a pair of keys—one public for distribution to other parties and the other private for decryption—are used for message encryption and decryption.

■ **Caution** While secure key generation is a boon provided by the Bouncy Castle API, it's not the end of the story. What your application does with its cryptographic keys is as important as how they're created; weak storage or exchange (such as sharing keys for symmetric ciphers over an unsecured network channel) defeats the purpose of having a good key-generation algorithm in the first place.

Creating Secure Commerce with Contactless Communications

Contactless communication—encompassing *near-field* communications such as RFID tags and bar codes—is becoming an important segment of the mobile market. With proper support for contactless communications, you can create many kinds of applications, such as swipe-to-purchase (using your mobile phone as a mobile wallet), mobile-purchase (scan a bar code and enable remote purchase), or comparison-shopping (scan a bar code to learn more about a product, including comparison prices). JSR 257 defines the Contactless Communication API, a suite of optional packages that provides support for both RFID tags (also known as *proximity tags*) and bar codes (also known as *visual tags*).

■ **Tip** As I write this, your best bet for a development environment that lets you work with the Contactless Connection API is the Nokia Near Field Communication (NFC) SDK, which you can find on the Web by going to `http://www.forum.nokia.com/main/resources/technologies/nfc/`.

The Contactless Communication API provides five packages, four of which may or may not be present in any given implementation of the API:

- `javax.microedition.contactless`: Always present and provides classes that let you discover near-field devices in the immediate vicinity
- `javax.microedition.contactless.ndef`: Provided if communications with cards supporting the NFC Data Exchange Format (NDEF) are supported
- `javax.microedition.contactless.rf`: Provided if communications with general RFID cards are supported
- `javax.microedition.contactless.sc`: Provided if communications with RFID cards meeting the ISO 144443 proximity standard are supported
- `javax.microedition.contactless.visual`: Provided if the implementation supports bar-code recognition and display

These packages rely on the presence of CLDC 1.1 or greater on the device, or equivalent support from another Java configuration.

Contactless communication devices come in a variety of flavors, but all are based on some wireless protocol typically built around RFID tags. The Contactless Communication API presently supports the following kinds of contactless devices:

- *NDEF-compliant devices*: Communication occurs through the exchange of `NDEFMessage` objects that possess individual `NDEFRecord` objects.
- *ISO 144443–compliant devices*: Communication occurs through a `Connection` subclass that permits the exchange of APDUs represented as arrays of bytes.
- *Generic RFID tags*: Communication with these devices occurs through a `Connection` subclass that lets you exchange vectors of commands.

Not every implementation of the Contactless Communications API supports all of these options.

When using the Contactless Communication API, your application typically performs one or more of the following operations:

- *Register for notifications when a contactless target becomes available*: You can do this using the MIDP push registry (see Chapter 14 for how to use the MIDP push registry) or the `javax.microedition.contactless.DiscoveryManager` class provided by the API.
- *Communicate with a contactless communications device using the appropriate* `Connection` *subclass provided by the Contactless Connection API*: You can do this when the device comes into range—reported either as an MIDP push or an event from the `DiscoveryManager`.
- *Capture the image using MMAPI, if you require visual tag recognition*: For details about how to do this, see Chapter 16. You pass the image you capture to a `VisualTagConnection` instance for recognition.
- *Create a stream of bytes to be encoded as a bar code and pass the bytes to a* `VisualTagConnection` *to generate an image, if you require visual tag generation*: Once the image is generated, present it to the user.

Discovering Contactless Targets

Contactless communication poses a challenge, in that your application doesn't know *when* it will encounter a target with which it can communicate. Your application may need to always be listening for a target (applications that confirm payments at point of

sale or those used for secure identification mechanisms generally fall into this category) or may only listen for a target while the application is active—for instance, once the user constructs a message to send to a target.

To always listen for a contactless target, your application simply needs to register this intent with the MIDP push registry. The API provides push notifications when NDEF-enabled or contactless targets come into proximity with the Java ME device. To register for a push notification when this occurs, you need to construct a URI indicating the type of NDEF records the application seeks, and place this URI in a field of the JAD file with the name `MIDlet-Push-`*n*.

The URI can specify whether your application should start when the Java ME device discovers a target containing either specific MIME-encoded data or a specific record type. In either case, the push registry URI begins with the protocol `ndef` and then continues with either a MIME type declaration or record type declaration, like this:

```
ndef:mime?name=text/x-uri
```

This push registry URI will launch your application whenever the Java ME device encounters an NDEF-enabled target bearing a URI (that is, a block of data with the MIME type `text/x-uri`). Here's another example:

```
ndef:rtd?name=urn:nfc:wkt:Sp
```

In this example, the first two fields of the URI—`urn` and `nfc`—indicate that the record is an NDEF record, while `wkt` indicates the namespace for the record. Finally, `Sp` indicates that the target being sought is a Smart Poster.

If your application uses the registry to receive notification of proximate targets, it should also implement the `javax.microedition.contactless.ndef.NDEFRecordListener` interface, which gives the platform a means to communicate data from the target to your application. This method takes a single argument, an `NDEFRecord` generated by the target. (I discuss these records in more detail in the next section.)

Once your application launches, you can also listen for targets by using the `DiscoveryManager` class, provided by the `javax.microedition.contactless` package. Required by all Contactless Communication API–compliant devices, this interface lets you enumerate which kinds of targets are supported, and it lets you register a listener when the Java ME device encounters a specific target. The listener must implement the `javax.microedition.contactless.TargetListener` interface, which defines a single method, `targetDetected`, that the API invokes when the device encounters a new target. The API passes this method an array of `TargetProperties` that describes the target the device has encountered. For example, Listing 15-8 registers a listener for NDEF-enabled targets.

Listing 15-8. *Registering a Listener for NDEF-Enabled Targets*

```
import javax.microedition.contactless.*;
import javax.microedition.contactless.ndef.*;
import javax.microedition.pim.*;
import java.util.Enumeration;
import javax.microedition.io.Connector;

public class NDEFExample implements TargetListener {
    private DiscoveryManager dm;
    NDEFTagConnection conn;

    public void registerForDiscovery() {
        TargetType[] targets = DiscoveryManager.getSupportedTargetTypes();
        boolean supported = false;
        for (int i=0; i<targets.length; i++) {
            if (targets[i].equals(TargetType.NDEF_TAG)) {
                supported = true;
            }
        }
        if (supported) {
            dm = DiscoveryManager.getInstance();
            try {
                dm.addTargetListener(this, TargetType.NDEF_TAG);
            }
            catch (ContactlessException e) { … }
        }
    }

    public void targetDetected(TargetProperties[] prop) {
        TargetProperties target = prop[0];
        String url = target.getUrl();
        try {
            conn = (NDEFTagConnection)Connector.open(url);
            if (conn != null) {
                readMessage();
            }
        }
       catch (IOException e) { }
    }
```

```
    public static void readMessage()
    {
    …
    }

    public static void writeMessage()
    {
    …
    }
}
```

It's important that registration begin with a determination of which target types are supported; you can determine which target types a specific implementation of the API supports by invoking the `DiscoveryManager`'s static `getSupportedTargetTypes` to obtain a list of supported types.

■ **Tip** Always enumerate the supported types! Some versions of the Contactless Communication API may support *additional* types of near-field devices beyond what I discuss here; see the documentation that accompanies a specific Java ME device and implementation of the API for details.

Once the code tests to ensure that the implementation supports the NDEF type, it registers a listener using the `addTargetListener` method. If you want to listen for multiple target types, you can do so using a single `DiscoveryManager`; you can either specify separate objects to respond to the `targetDetected` event or handle the detected target type within the `targetDetected` method.

After registration, the API implementation invokes your listener's `targetDetected` method when the device encounters a target of the appropriate type; at this point, your code should begin communicating with the device. This may be a place where your application needs to launch a separate thread, especially if the communications exchange consists of more than just a command or two.

Communicating with Contactless Targets

Unsurprisingly, communication with targets is via the ubiquitous GCF; the means to communicate with various targets are embodied in `Connection` subclasses, as you see in Figure 15-3.

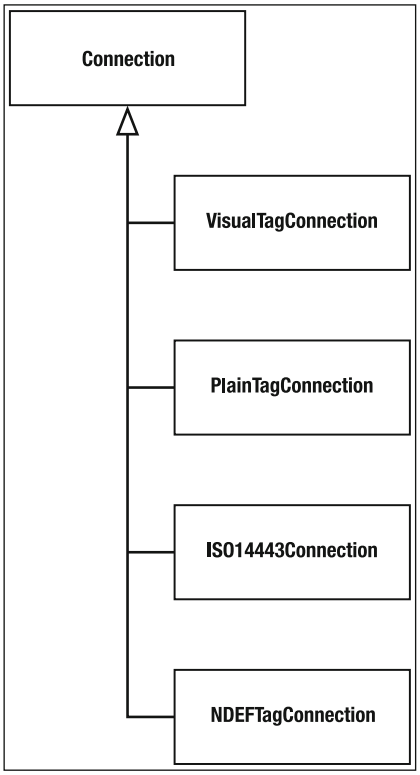


Figure 15-3. *Supporting classes for communication with contactless targets*

Unfortunately, implementing `Connection` is where uniformity between the various subclasses ends. The API takes a message-oriented approach to communicating with contactless targets, but the methods differ between the types of targets:

- NDEF-enabled devices use the `javax.microedition.contactless.ndef.NDEFTagConnection` class to exchange `NDEFMessage` instances via the connection's `readNDEF` and `writeNDEF` methods.
- ISO 14443–enabled devices use the `javax.microedition.contactless.sc.ISO14443Connection` class and its `exchangeData` method to exchange raw arrays of bytes.
- Generic RFID devices use the `javax.microedition.rf.PlainTagConnection` class. Instances of this class exchange vectors of Java objects using the `tranceive` method. These Java objects are defined by extensions to the API that encapsulate application-specific data.
- Visual tags can be encoded and decoded using the `javax.microedition.contactless.visual.VisualTagConnection` class, which actually doesn't connect to anything. (I say more about this class in the next section.)

NDEF defines *messages* consisting of one or more *records,* each in a lightweight binary message format that consists of a payload type, optional payload identifier, and payload itself. For example, to read any pending NDEF records, you might write the code shown in Listing 15-9.

Listing 15-9. *Reading Pending NDEF Records*

```
public static void readMessage()
{
    try
    {
        NDEFMessage message = conn.readNDEF();
        if(message == null)
        {
            return;
        }
        NDEFRecord[] records = message.getRecords();
        if(records == null)
        {
            return;
        }
        System.out.println("Got "+records.length+" records.");
        for (int i = 0; i < records.length; i++)
        {
            NDEFRecordType type = records[i].getRecordType();
            System.out.println("id: " + new String(records[i].getId()) );
            System.out.println("type: " + type.getName() );
            System.out.println("payload: " + new String(records[i].getPayload()));
        }

    } catch (Exception e) { … }
}
```

This code walks the list of returned records, printing the ID and payload of each message. Messages may also bear a type, available by invoking the record's `getRecordType` function, which returns an instance of `NDEFRecordType`. This type defines several record types, including `EMPTY`, `NFC_FORUM_RTD`, `MIME`, `URI`, `EXTERNAL_RTD`, and `UNKNOWN`. A common type is the URI type, which indicates that the payload of the message is a URL, such as one offered by a Smart Poster device.

If you need to write records to the NDEF target, you can create a new `NDEFRecord` and use the `NDEFTagConnection`'s `writeNDEF` method. When you define a new `NDEFRecord`, you must supply the record's type, ID, and payload, as shown in Listing 15-10.

Listing 15-10. *Defining a New NDEF Record*

```
public static void writeMessage()
{
    try {
        PIM pim = PIM.getInstance();
        ContactList cl =
            (ContactList)pim.openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY);
        Enumeration items = cl.items();
        Contact item = (Contact) items.nextElement();
        String name = item.getString(Contact.FORMATTED_NAME,0);
        NDEFRecord[] records = new NDEFRecord[1];
        records[ 0 ] = new NDEFRecord(
            new NDEFRecordType(NDEFRecordType.UNKNOWN, “name”),
            new String("F_M_NAME").getBytes(),
            name.getBytes());
        NDEFMessage message = new NDEFMessage(records);
        conn.writeNDEF(message);
    } catch (Exception e) { … }
}
```

This code transmits the first name in the PIM contacts list. First, it needs to ascertain that name, which it does by using the PIM API (see Chapter 7) to obtain an instance of the PIM manager. It uses the PIM manager to obtain the contact list and then uses the enumeration the contact list provides to obtain the name stored in the first record of the contact. Then the code creates a new NDEF record array, consisting of a single record. This record is assigned a new `NDEFRecord`, of type unknown, bearing the ID `F_M_NAME` and the formatted name obtained from the contact list. A new message is created using the record array and then written via the connection.

Reading and writing to other contactless targets such as ISO 14443 is similar, although instead of using separate read and write methods that the connector provides, you exchange messages directly, as shown in Listing 15-11.

Listing 15-11. *Exchanging Messages Directly*

```
import java.io.IOException;
import javax.microedition.contactless.*;
import javax.microedition.contactless.sc.*;
import javax.microedition.io.Connector;
```

```
public class ISO14443Example implements TargetListener {
    byte[] commandAPDU = ...;
    public ISO14443Example () {
        try {
            DiscoveryManager dm = DiscoveryManager.getInstance();
            dm.addTargetListener(this, TargetType.ISO14443_CARD);
            dm.addTransactionListener(this);
        }
        catch (ContactlessException e) { … }
    }

    public void targetDetected(TargetProperties[] properties) {
        TargetProperties target = properties[0];
        Class[] classes = target.getConnectionNames();
        for (int i=0; i<classes.length; i++) {
        try {
            if (classes[i].equals(Class.forName(
                "javax.microedition.contactless.sc.ISO14443Connection")
                )) {
                    String url = target.getUrl(classes[i]);
                    ISO14443Connection smc =
                        (ISO14443Connection)Connector.open(url);
                    byte[] responseAPDU = smc.exchangeData(commandAPDU);
                    // do something with the response
                }
            }
            catch (Exception e) { … }
        }
    }
}
```

The detection process for an ISO 14443 target (or other RFID target, for that matter) is exactly the same as for an NDEF-enabled one; simply specify a `TargetType` of `ISO14443_CARD` to the `DiscoveryManager`'s `addTargetListener` interface. Once a target is detected, the platform invokes the registered listener `targetDetected`, which can then exchange APDU packets with the target.

If this interface looks familiar, it should; that's because ISO 14443 devices are close kin to the wired smart cards supported by the optional SATSA API I discussed previously in this chapter.

■ **Caution** You can only have a single connection open to a contactless target, due to hardware limitations of most contactless target devices.

Recognizing and Generating Visual Tags

Visual targets—a fancy name for bar codes, and you'll often see the interface specification refer them to *tags*—provide an interesting avenue for mobile e-commerce applications, because they are ubiquitous on product packaging today. In turn, this means that readers for these targets are also commonplace, enabling applications such as the transfer of credentials to readers for paperless tickets.

The support for visual tags in the Contactless Connection API is somewhat different than for other contactless targets, because there might be no automated discovery process. While your application can register with the `DiscoveryManager`, you can also initiate manual scanning of a visual tag. If the user wants to scan a visual tag, it's up to her to see the bar code, start your application, and have the application capture the bar-code image using the MMAPI interface and present the image to the Contactless API for recognition. In a similar vein, you can encode arbitrary data into a bar code and display it on the screen, so the user can present a bar code to a bar-code scanner.

Bar codes come in different flavors, called *symbologies*. Some, such as European Article Number/Universal Product Code (EAN/UPC), are one-dimensional, containing a single stripe of information. Other, newer bar codes, such as Quick Response Code (QR Code), are two-dimensional squares or rectangles of information. These two-dimensional bar codes can encode more information, and applications often use them to carry URLs or other information. When using the visual tag interface, you first need to ascertain whether the symbology you desire is supported by the implementation. You determine the list of supported symbologies by invoking the static `SymbologyManager.getReadSymbologies` method, which returns an array of strings naming the supported symbologies. Table 15-1 shows a list of potentially supported symbologies.

In addition to being symbology-agnostic, the Contactless Communication API is image format–agnostic. The `SymbologyManager` keeps a list of supported image classes that the implementation can decode or encode; you can obtain these lists through the static method `SymbologyManager.getImageClasses`. This method returns an array of supported `Class` instances; you can then peruse this list to see if the implementation supports a specific image type.

Once you ascertain support for your desired symbology and image type, it's time to perform the encoding or decoding. You do this using a `VisualTagConnection` object, which you obtain by opening a `Connector` to the URI `"vtag://"`. The resulting instance of `VisualTagConnection` can perform the necessary encoding or decoding. It's kind of odd to use a `Connection` subclass to do something besides establish a connection, but that's how the Contactless Connection API specifies the interface; it at least provides conceptual parity with the other contactless interfaces.

Table 15-1. *Symbologies Typically Supported by the Contactless Communication API*

Symbology Name	Body Responsible for the Standard
aztec-code	ANSI/AIM B13 ITS/97/002
code-16k	ANSI/AIM BC7/1995
code-39	ANSI/AIM BCI-1995, ISO/IEC 16388
code-49	ANSI/AIM BC6-1995
code-93	ANSI/AIM BC5-1995
code-128	ANSI/AIM BC4-1999, ISO/IEC 15417
codebar	ANSI/AIM BC3-1995
data-matrix	ISO/IEC 16022
ean-upc	ISO/IEC 15420
interleaved-2-of-5	ANSI/AIM BC2-1995, ISO/IEC 16390
maxicode	ANSI/AIM BC10, ISO/IEC 16023
pdf417	ISO/IEC 15438
qr-code	AIM ITS/97/001, ISO/IEC 18004

Listing 15-12 shows a class that encapsulates the encoding and decoding of QR Codes.

Listing 15-12. *Encoding and Decoding QR Codes*

```
import java.io.IOException;
import javax.microedition.lcdui.*;
import javax.microedition.io.Connector;
import javax.microedition.contactless.ContactlessException;
import javax.microedition.contactless.visual.*;

public class ProcessVisualTag {
    private final String sym ="qr-code";
    private Class imageClass;
    private byte[] data;
    private Image image;

    public ProcessVisualTag(Object im) {
        image = (Image)im;
        imageClass = im.getClass();
        decode();
    }
```

```
public ProcessVisualTag(byte[] d) {
    data = d;
    imageClass = new String("javax.microedition.lcdui.Image");
    encode();
}

public byte[] getData() {
    return data;
}

public Image getImage() {
    return image;
}

private boolean isSupported() {
    boolean supportedSym = false;
    boolean supportedImage = false;
    try {
        String[] symbologies = SymbologyManager.getReadSymbologies();
        for (int i=0; i<symbologies.length; i++) {
            if (symbologies[i].equals(sym)) {
                supportedSym = true;
                break;
            }
        }
        Class[] images = SymbologyManager.getImageClasses();
        for(int i=0; i<images.length; i++) {
            if (images[i].equals(imageClass) ) {
                supportedImage = true;
                imageClass = images[i].getClass();
                break;
            }
        }
    }
    catch (Exception e) { return false; }
    return supportedSym && supportedImage;
}
```

```
    private void decode() {
        try {
            if ( isSupported() ) {
                VisualTagConnection conn =
                    (VisualTagConnection)Connector.open("vtag://");
                 data = conn.readVisualTag(image, imageClass, sym);
                 conn.close();
            }
        }
        catch (IOException ioe) { ... }
        catch (VisualTagCodingException ce) { ... }
    }

    private void encode() {
        try {
            if ( isSupported() ) {
                ImageProperties props =
                    SymbologyManager.getImageProperties(sym);
                VisualTagConnection conn =
                    (VisualTagConnection)Connector.open("vtag://");
                 image = (Image)conn.generateVisualTag(data, imageClass, props);
                 conn.close();
            }
        }
        catch (ContactlessException ce) {…}
        catch (IOException ioe) {…}
        catch (VisualTagCodingException ce) {…}
    }
}
```

On creation, this helper accepts either an image object (such as that obtained using the MMAPI, as I show you in Chapter 16) or an array of bytes representing a byte stream, and it creates the corresponding representation as either an array of bytes or an encoded QR Code. If you obtain an encoded QR Code, you will receive the data as an instance of the `Image` class. The `isSupported` method handles the discovery of supported symbologies and image representations, querying the `SymbologyManager` for its list of supported symbologies and image classes. (When the code finds a matching image class, I copy it to the `imageClass` instance variable, because the method for generating a visual tag takes a `Class` instance, not the name of a class.)

The `VisualTagConnection` interface provides two methods: `readVisualTag` and `generateVisualTag`. The `readVisualTag` (which you see in the `decode` method in Listing 15-12) method is straightforward, accepting the source image, the class of the source image, and the encoding symbology; the bytes generated from a successful scan are returned as an array of bytes. This method can throw an `IOException` or a `VisualTagCodingException`, which indicates that the coding operation failed.

The `generateVisualTag` (which you see in the `encode` method in Listing 15-12) method is the encoding counterpart to `readVisualTag`, and it accepts the data to encode, the class to which the data should be encoded (presumably an `Image` or subclass), and a set of properties describing symbology-specific properties of the encoding operation. You should always get these image properties for a specific symbology from the `SymbologyManager` by invoking its `getImageProperties` method; the resulting properties define things like the dimensions of the generated bar code as well as the pixel pitch.

Wrapping Up

The Java ME platform provides a modicum of security-related interfaces to application developers, like the inclusion of HTTPS for secure network transactions. However, there's more to crafting a secure application than simply encrypting network traffic; a secure application may need to defend itself from threats including unauthorized use or access to its private data.

Fortunately, solutions to security challenges exist, providing ciphers for data hiding, digests and signatures to defend data against tampering and prove origination of a message, and certificates to provide identity. The Java community has responded to the need by implementing these security tools in a number of ways, including SATSA that JSR 177 defines; the open source Bouncy Castle API, which provides a comprehensive suite of security and cryptographic interfaces; and higher-level building blocks to build secure commerce applications, like the Contactless Communication API that JSR 257 defines.

The SATSA optional interfaces available on some devices provide various hardware and software solutions to security problems, including APIs to access cryptographic smart cards, infrastructure for the secure management of public keys, and a limited subset of the JCA for encrypting and decrypting data. Using the SATSA optional interfaces, your application can access a variety of smart cards using the GCF at either the level of individual bytes or Java RMI if the device supports Java smart cards. Your application can also use this API to perform encryption and decryption with symmetric ciphers.

The open source Bouncy Castle API provides a comprehensive suite of cryptographic solutions to security problems, bringing with it support for many different kinds of ciphers and message digests. It also provides implementations for other data representations, including ASN.1 and S/MIME, as well as support for OpenPGP encoding and decoding. The interface is similar to that defined by SATSA for common operations such as encryption, decryption, and the creation of message digests, and its clean-room

implementation in pure Java makes it an ideal choice for inclusion in any application that needs to use its features while remaining portable to hardware that may or may not support SATSA.

Today's mobile applications leverage security solutions in a number of ways, but electronic commerce is playing an increasing role in driving secure technologies to mobile devices. To support secure commerce applications, the optional Contactless Communication API provides the means to communicate with a number of wireless data devices, such as near-field wireless devices and bar codes. The API provides support for both proximity cards at a low level (where you exchange individual bytes) as well as the NDEF, and it also lets you decode or generate bar-code images.

Security is a big topic, and this chapter has only scratched the surface. It's not like cooking with spices; while you can dump a handful of peppers into a dish to make it taste spicy, you can't just dump a handful of security APIs into your application and make it secure. (Actually, they're more alike than you think: the resulting dish is as likely to be as inedible as your application is insecure!) Designing a secure application requires care and forethought, and if you're working to that end, be sure to consult one or more of the excellent Internet and book references mentioned in this chapter for more details.

CHAPTER 16

Rendering Multimedia Content

Today's consumers increasingly demand the rich multimedia experience to which they're accustomed from their personal computers or other devices, including mobile wireless terminals and portable media players. At the same time, both to sate this demand and distinguish a product from its competition, manufacturers increasingly invest money in developing rich multimedia interfaces for their product. While pundits may argue that much of this investment has been gratuitous—little of the glitz and glamour of a cell phone's user interface today is truly necessary to make a call, for example—much of it has not. Multimedia applications, from pedestrian audio and video playback to sophisticated data-visualization applications, can both differentiate products and add value. Through standardization efforts such as JSR 135, which defines the MMAPI, and JSR 287, which defines support for Scalable 2D Vector Graphics, Java ME provides a wealth of interfaces that enable you to build multimedia-rich applications.

In this chapter, I show the interfaces these JSRs define. I begin by discussing the MMAPI, because its support for audio and video makes it an important component of many applications today. I explain how the MMAPI is organized and how it uses the Java runtime and native support from the host hardware to render content from the device's local storage and the network. Next, I turn attention to Java ME's support for SVG—an exciting standard under development by the W3C and widely supported by browsers as well as mobile devices. I show you how to render SVG images and animations using the classes provided by the optional Scalable 2D Vector Graphics API (which I'll just call the SVGAPI for short), as well as how to build Java ME applications that leverage SVG content for parts of their user interface. After going over the basics of all the multimedia-supporting classes at your disposal, I present an example MIDlet that plays audio, video, and SVG content and lets you interface with a device's camera to capture images.

Introducing the MMAPI

From relatively early in the history of mobile Java, hardware vendors and Sun worked together to explore how Java should support the rendering of multimedia content. However, as participants in the Java community were hammering out the details of the MMAPI, the hardware available could barely render digital audio, let alone video. Everyone soon recognized the important role that multimedia support would play for many Java ME applications.

The MMAPI features a modular API that enables you to both create your own simple audio media through the specification of frequency and tone data as well as render multimedia audio and video using a variety of codecs. Because the MMAPI is extensible, hardware manufacturers can add new codecs and transports so that you can use the same interfaces to render new data formats as they become available.

■ **Note** Limitations of space prevent me from thoroughly covering everything you can do with the MMAPI. If you find yourself wanting to learn more about the MMAPI after you read this chapter, I encourage you to read *Pro Java ME MMAPI: Mobile Media API for Java Micro Edition* by Vikram Goyal (Apress, 2006), which goes into far more detail about how you can use the MMAPI in your Java ME applications.

Understanding Basic Multimedia Concepts

To use the MMAPI successfully, you must understand three key concepts about how it divides the responsibilities of media rendering. The first concept has to do with how creators and distributors package and deliver multimedia content. Content creators (be they major companies or individuals posting to social sites like YouTube) use specific *media types* when exchanging multimedia content. Today's media types are typically sophisticated file formats that act as *containers* for highly compressed streams of audio-visual data. For example, the popular Moving Picture Experts Group Version 4 (MPEG-4) standard defines not just one but a suite of audio and video coding formats; an MPEG-4 file may have an audio stream in Advanced Audio Coding (AAC) synchronized with a H.263 video stream, or it may be PureVoice audio synchronized with an MPEG-4 video stream, or it may be something else altogether. Thus, when specifying what kinds of multimedia their hardware can render, device vendors typically specify the information in terms of both container formats *and* codecs. Typically, most mobile devices today rely on a complex combination of hardware through digital signal processors (DSPs), dedicated integrated circuits, and so forth, as well as software to implement the codecs necessary for rendering multimedia. The implementation of a specific MMAPI stack interconnects with this hardware and software to provide highly efficient mechanisms for rendering a variety of multimedia formats.

Caution The boundary between container file formats and data file formats is often a blurry one, especially when many encodings are themselves containers. When working with content providers, it's not enough to say something like, "I'll take that as an MPEG-4 file, please," the way you might ask for a PNG image. Worse, the coding schemes supported by different devices are often different, so you may find that an MPEG-4 file that plays on one device doesn't play on another, because the streams of data *inside* the file are coded using a compatible codec on the first device (say, in AAC on a device that supports AAC) and incompatible on the second. Later, I show you how to interrogate a device's MMAPI implementation to determine just what codecs are available.

Second, content providers can deliver multimedia in files—an all-at-once process in which the device stores an entire file (say, a song or video) on the device's file system—or stream data using protocols such as the Real Time Streaming Protocol (RTSP). Most interfaces to multimedia systems treat the source of the data as a discrete entity, aptly calling it the *data source*. A data source hides all the details of how the device obtained the encoded data; it may have fetched the data from a file or over the network using any one of a number of open or proprietary technologies.

Third, the low-level notion of a codec is usually at the wrong level of abstraction for the average application developer. Working at the application layer, it's easiest to think of a *player*: an engine that, given a data source and some information about the how the data coming from that data source was encoded, can render the data as an audio or video stream. Players typically either offer interfaces directly for managing rendering, or, on some platforms like that provided by the MMAPI, provide an interface to obtain one or more *controllers* that can control a specific player's rendering of the data from the data source.

Note Experienced readers will doubtless see parallels between the MVC pattern and the source-player-controller organization of many multimedia APIs. This is not a coincidence; the division of responsibilities that MVC provides fits well with the work necessary to render a multimedia stream, at least from the perspective of you and me as application developers.

Finally, while you're likely most interested in rendering multimedia from a data source such as a web service or file, it's important to realize that today's devices may offer their own data sources. A camera phone, for example, has an imaging sensor; many implementations of the MMAPI permit you to treat that imaging sensor as a data source from which you can render video and even capture images. When thinking about multimedia applications, it's important to remember that the source-player-controller division of responsibilities is a general one and can be applied to almost anything that represents a stream of multimedia data.

Understanding the Organization of the MMAPI

Figure 16-1 shows the key classes that the MMAPI provides. In the diagram, the `DataSource` class plays the role of the data source, the `Player` interface plays the role of the player, and the various implementors of the `Control` interface play the role of controllers in the division of labor required to render multimedia on Java ME devices. Three packages contain all of these classes: `javax.microedition.media`, `javax.microedition.media.control`, and `javax.microedition.media.protocol`.

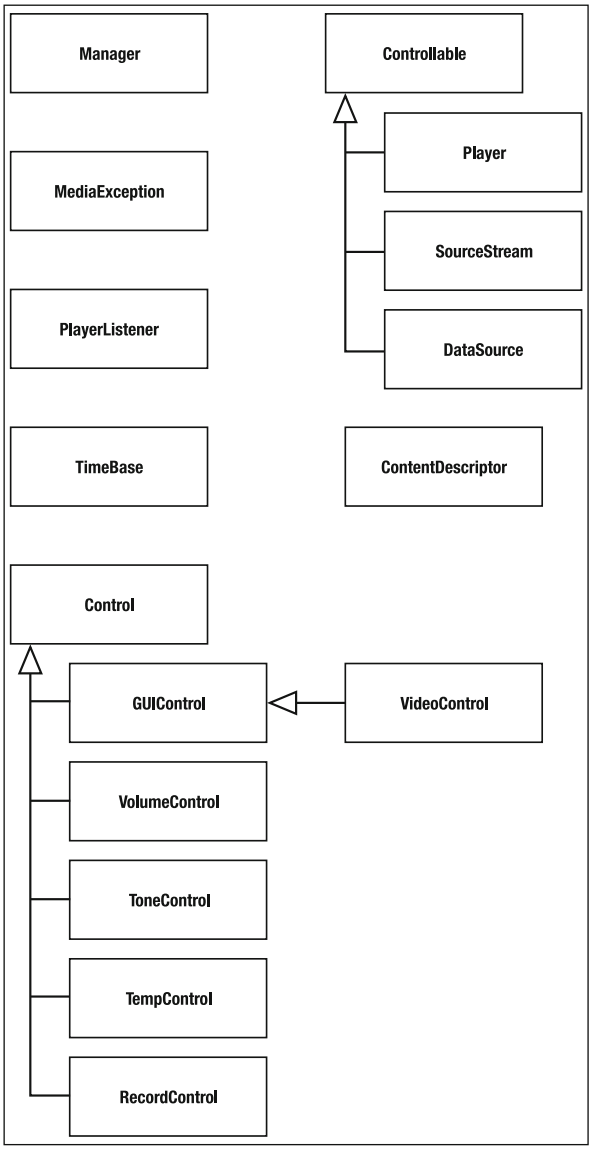


Figure 16-1. *Key classes the MMAPI provides*

The MMAPI provides the following classes and interfaces, among others:

- `Manager`: The `Manager` class provides an access point for obtaining system-dependent resources such as implementations of the `Player` interface. `Manager` methods are static, so there's no need to obtain an instance of the class.
- `Player`: The `Player` interface provides the methods for interacting with a piece of multimedia. Subclasses implementing `Player` encapsulate specific rendering codecs such as an MP3 audio player.
- `Control`: The `Control` interface defines some media processing functions. `Control` implementations interact with objects implementing the `Controllable` interface, such as `Player` instances to permit programmatic and user control of media. Examples include the `VolumeControl`, which permits you to adjust the playback volume, and the `VideoControl`, which permits you to control the visibility and placement of the video on the display.
- `DataSource`: Implementations of the `DataSource` abstract class provide media to instances of `Player`. They provide a container indicating a media object's content type and location as well as streams of source data through instances of `SourceStream` that provide random access to a single media stream.
- `ContentDescriptor`: Instances of the `ContentDescriptor` class bear a string indicating the type of a specific `DataSource`.
- `TimeBase`: The `TimeBase` interface represents a constantly ticking source of time. Your application can share one `TimeBase` instance between multiple players to synchronize multiple streams of media.
- `PlayerListener`: The `PlayerListener` interface describes the asynchronous events a `Player` instance can provide to applications. You can implement this interface to listen for notifications about media rendering.
- `MediaException`: MMAPI implementations use instances of the `MediaException` class to signal errors.

■ **Note** The `Control` interface has many subinterfaces, and I don't discuss all of them here. For a thorough explanation of what each does, consult JSR 135 or the Javadoc that accompanies the MMAPI implementation.

Using the MMAPI to render multimedia is simple; you must follow these four steps:

1. Create a `Player` instance using the `Manager` class, and reference a path to a specific piece of media.
2. Permit the `Player` instance to obtain any necessary hardware resources required to render the media.
3. Obtain any necessary `Control` instances to control or configure the `Player` instance.
4. Play the media.

Because the MMAPI implementation typically wraps dedicated hardware resources (such as RAM, dedicated DSPs, or dedicated coprocessors) that other processes on the device might require, it's not enough to instantiate a `Player` instance. Instead, a `Player` instance can be in one of five states, giving you programmatic control over the potentially time-consuming process of obtaining necessary system resources (and permitting you to better share those scarce resources with other applications on the device). Figure 16-2 shows how these states interact.

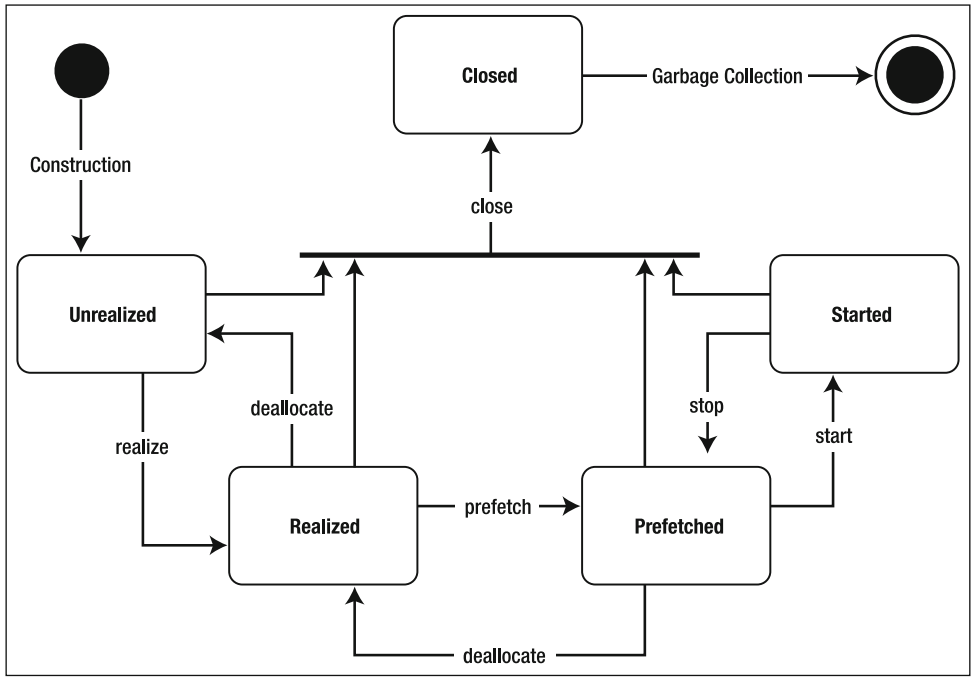


Figure 16-2. *The states of a Player instance*

These states are

- *Unrealized*: An unrealized `Player` instance does not have the information it needs about the resources it must acquire in order to function. Many `Player` methods, including `getContentType`, `setTimeBase`, `getTimeBase`, `setMediaTime`, `getControls`, and `getControl`, will fail if invoked on a `Player` in this state, and the `Player` will throw an `IllegalStateException`.

- *Realized*: An unrealized player moves to the realized state when you invoke its `realize` method. This transition can be a lengthy process as the `Player` instance obtains system resources and works with the media content to determine its content type or perform other operations. You can stop the realization process by invoking the instance's `deallocate` method, but you must do so *before* the `Player` instance finishes transitioning to the realized state; once realized, a `Player` instance cannot return to the unrealized state.

- *Prefetched*: A realized player may still need to do additional time-consuming processing such as fill media buffers from the network or obtain exclusive resources such as a hardware codec. To do this, you invoke the `Player` instance's `prefetch` method.

- *Started*: Once a `Player` instance has been realized and its contents and resources have been prefetched, you can start playback by invoking the `Player` instance's `start` method.[1] This causes the `Player` instance to transition to the started state. While in the started state, the instance runs and renders multimedia data until you stop it by invoking `stop` or until it runs out of data to render. Invoking `stop` on a started `Player` instance returns it to the prefetched state.

- *Closed*: From any other state, you can force a `Player` instance to release its resources and move to the closed state by invoking its `close` method. Once placed in this state, you must not attempt to use the instance again; it's ready for reclamation by the garbage collector, and all you should do is set any references to it to `null`.

As they say, the devil's in the details, so let's look at some actual code that renders multimedia and controls its playback.

1. Oddly, you can usually bring a realized `Player` instance directly to the started state by invoking its `start` method. A lot of the tutorial examples you see on the Internet—including examples from Sun—do this, but it's counter to the documentation that the MMAPI JSR provides, so I don't recommend it.

Starting the Rendering Process

As a factory for `Player` instances, the `Manager` class has a straightforward set of interfaces. You can play single tones (something I discuss further in the "Playing Individual Tones" section later in this chapter), obtain an instance of `Player` for a specific piece of media, list the supported content types and protocols, and get the system's time base used for rendering all media. The method you use most often, of course, is the `createPlayer` method to obtain a `Player` instance.

`Manager` actually has three different `createPlayer` methods. The easiest one to use is the one that takes an `InputStream` and a `String` indicating the MIME type of the media, but it limits you to playing multimedia to which you can get an `InputStream`, such as a static resource in your JAR file. For example, you might write the code in Listing 16-1 to create, realize, prefetch, and start an MP3 audio player using a local resource:

Listing 16-1. *Pseudocode for Playing an MP3 Audio File*

```
Player player;
try {
    InputStream in = getClass().getResourceAsStream("/res/sound.mp3");
    player = Manager.createPlayer(in, "audio/mp3");
    player.realize();
    player.prefetch();
    player.start();
} catch (MediaException e) {}
  catch (IOException e) {}
```

Of course, you could use any other means to open an `InputStream` to your media source; for example, you might want to use the FCOP I describe in Chapter 7 to play user-loaded media files on the file system.

Often, though, you may want to specify a different source for the media, such as a network source or another component of the device, such as an imaging sensor. The `Manager` class also supports the notion of a *locator*, which is a lot like the URLs you pass to the GCF. A locator is a `String` you pass to `Manager.createPlayer`. The `String` must contain the following parts:

```
scheme://scheme-part
```

Specifically, `scheme` is the name of a protocol, such as `http` for HTTP, `rtp` for the Real-Time Transfer Protocol (RTP), or `rtsp` for RTSP. The `scheme-part` is the location of the media in a scheme-specific way, such as the host and path of a resource on the Web. Table 16-1 shows some commonly supported schemes and their purpose.

■ **Caution** Not all devices support all schemes. For example, RTSP support is a relative latecomer to the MMAPI; as I write this, only a few devices include support for it.

Table 16-1. *Some Commonly Supported MMAPI Schemes*

Scheme Name	Purpose	Note
`capture`	Captures live media from a device sensor	The `scheme-part` indicates the capture device.
`http`	Fetches via HTTP	It usually downloads the entire file *before* rendering.
`rtp`	Streams via RTP	The `scheme-part` indicates the server address, port, and content type.
`rtsp`	Streams via RTSP	The `scheme-part` indicates the server address, port, and resource from which to stream.

Listing 16-2 shows some examples of syntactically valid locators. However, just because a locator is syntactically correct does *not* mean that the device has the necessary protocol and codec support to play the media that the locator specifies.

Listing 16-2. *Some Valid Locators*

```
http://www.noplace.com/loon-hoot.mp3
http://www.noplace.com/loon-display.3gp
capture://video
capture://audio
capture://radio
rtp://www.noplace.com:1224/audio
```

Under the hood, when you pass a locator to `Manager.createPlayer`, it sets up whatever protocol engine it needs after parsing the locator. It then creates a `DataSource` to pass media data from the source to the player. If you're looking to provide support for a protocol that isn't supported by the MMAPI implementation on a specific device, you can write a `DataSource` subclass that implements your protocol and use it with `Manager.createPlayer`, like you see in the pseudocode in Listing 16-3.

Listing 16-3. *Using a Custom DataSource with the MMAPI Manager*

```
Player player;
try {
    DataSource ds = (DataSource)new CleverStreamingDataSource(
        "proto://noplace.com/audio.mp3" );
    player = Manager.createPlayer(ds);
    player.realize();
    player.prefetch();
    player.start();
} catch (MediaException e) {}
  catch (IOException e) {}
```

Implementing `CleverStreamingDataSource` as a subclass of `DataSource` permits the `Manager` to create a player that connects to your data source for its data.

■ **Tip** Writing a custom `DataSource` is no easy task and is beyond the scope of this book. For more details, I suggest you begin with Vikram Goyal's article on Java ME content streaming, "Experiments in Streaming Content in Java ME," at `http://today.java.net/pub/a/today/2006/08/22/experiments-in-streaming-java-me.html`. In particular, take a look at the "Create a custom DataSource" section.

Regardless of how you create and start your `Player` instance, you will want to do so in a separate thread from the main application thread, because it's a time-consuming affair. I show you how to do this in practice later in this chapter, in the "Putting the MMAPI and the SVGAPI to Work" section.

Once you're done rendering media, at some point you will want to stop playback and reclaim the resources the `Player` instance has consumed. Listing 16-4 shows the usual process.

Listing 16-4. *Stopping Media Playback and Reclaiming Resources*

```
if (player != null)  {
    player.stop();
    player.close();
    player = null;
}
```

Interestingly, the `Player`'s `stop` method doesn't actually stop playback; it pauses it. That means that you can implement a user-initiated pause by invoking a `Player` instance's `stop` method, and you can restore playback where it was paused by invoking its `play` method again.

The MMAPI is very modular; to meet the goal of being available on the widest variety of platforms, not all features are available in all implementations. You can get a list of supported content types by invoking `Manager.getSupportedContentTypes`, or you can get a list of the supported protocols for each content type by passing a supported content type to `Manager.getSupportedProtocols`. Listing 16-5 enumerates the supported types and protocols on a given device, writing each to the debug console.

Listing 16-5. *Enumerating Supported Media and Protocol Types*

```
private void showSupportedMedia() {
    String[] contentTypes = Manager.getSupportedContentTypes(null);
    for (int i=0; i<contentTypes.length; i++) {
        String protocols[] = Manager.getSupportedProtocols(contentTypes[i]);
        for (int j=0; j<protocols.length; j++) {
            String s = contentTypes[i] + ":" + protocols[j];
            System.out.println(s);
        }
    }
}
```

In addition to determining whether a given MMAPI implementation supports a specific content type or protocol, you may need other information, such as whether the target device supports playing multiple audio channels simultaneously (audio mixing), whether the platform can capture audio or video, and so forth. The MMAPI defines a set of system properties whose values you can obtain using `System.getProperty`; these are shown in Table 16-2.

Table 16-2. *Properties Describing a Target Device's MMAPI Implementation*

Property	Returned Value
`microedition.media.version`	The version of the MMAPI specification implemented
`supports.mixing`	`true` if the device supports mixing; `false` otherwise
`supports.audio.capture`	`true` if the device supports audio capture; `false` otherwise
`supports.video.capture`	`true` if the device supports video capture; `false` otherwise
`supports.recording`	`true` if at least one returned `Player` type supports recording; `false` otherwise
`audio.encodings`	A string specifying the supported capture audio formats
`video.encodings`	A string specifying the supported capture video formats
`video.snapshot.encodings`	A string specifying the supported video snapshot formats for `VideoControl.getSnapshot`
`streamable.contents`	A string specifying content types that can be played by a `Player` instance as it receives data

Controlling the Rendering Process

For all but the simplest of media players, you will want to have additional control over media playback. While the `Player` interface itself lets you start and stop playback, you can do a lot more by obtaining instances of specific `Control` subclasses that let you change the `Player`'s behavior. To get an instance of a specific kind of control, you pass the name of the desired class (with the package name, if it's not `javax.microedition.media.control`) to the `Player`'s `getControl` method, like this:

```
VolumeControl vc = (VolumeControl)player.getControl("VolumeControl");
```

Each `Control` subclass has different methods, reflecting the kind of control over the media rendering process it offers. Table 16-3 describes some `Control` subinterfaces that come in handy when writing multimedia applications. Of course, not every subinterface is available for a `Player` instance rendering every media type; it would make little sense, for example, to be able to obtain a `VideoControl` for an audio file, or a `TempoControl` for a video file.

Table 16-3. *Implementors of Control You Can Use to Control Media Rendering*

Interface	Purpose	Example Methods

`FramePositioningControl`	Seek to a specific video frame	`seek`
		`skip`
		`mapFrameToTime`
		`mapTimeToFrame`

`GUIControl`	Provide an object to render the media to the GUI	`initDisplayMode`
`MetaDataControl`	Obtain metadata in a media file	`getKeys` `getKeyValue`
`MIDIControl`	Controlling an internal synthesizer	various

`PitchControl`	Shift pitch of synthesized or sampled audio without changing playback speed	`getMaxPitch`
		`getMinPitch`
		`getPitch`
		`setPitch`

`RateControl`	Shift playback rate	`getMaxRate`
		`getMinRate`
		`getRate`
		`setRate`

`RecordControl`	Record media from a player	`commit`
		`setRecordLocation`
		`setRecordStream`
		`startRecord`
		`stopRecord`

Interface	Purpose	Example Methods
StopTimeControl	Set a preset stop time	getStopTime setStopTime
TempoControl	Change tempo in synthesized audio	getTempo setTempo
ToneControl	Replay monochannel tone sequence	setSequence
VideoControl	Control video display	getSnapshot setDisplayFullScreen setDisplayLocation setDisplaySize setVisible

The GUIControl and its subinterface VideoControl deserve special attention, because they're how you get the video data from a multimedia source like a file or the device camera and display it on the screen. The GUIControl interface specifies a single method—initDisplayMode—which you use to obtain an object that can draw on a Displayable object. VideoControl extends this interface, providing you with methods to display video full-screen, specify the location of video playback, and so forth. Listing 16-6 shows how you use a VideoControl instance in conjunction with a Player instance to play a video clip.

Listing 16-6. *Playing a Video Using VideoControl and Player Instances*

```
Player player;
Form viewer;
VideoControl vc;

try {
    InputStream in = getClass().getResourceAsStream("/res/video.3g2");
    player = Manager.createPlayer(in, null);
    vc = (VideoControl) player.getControl("VideoControl");
    player.realize();
    player.prefetch();
    if (vc != null) {
        Item vi = (Item) vc.initDisplayMode(
            VideoControl.USE_GUI_PRIMITIVE, null);
        viewer.append(vi);
    }
    player.start();
} catch (MediaException e) {}
  catch (IOException e) {}
```

Personally, I find it a little creepy that you ask a `Control` for an object to place on the display, but that's how this works: the `VideoControl` instance's `initDisplayMode` takes a flag and an `Object` as an argument, and returns a visible component for the display. You pass the `VideoControl.USE_GUI_PRIMITIVE` value to indicate that the instance should return an instance of an object in the GUI hierarchy for the device. This example runs on the MIDP platform atop the CLDC, so the object that `initDisplayMode` returns is actually a subclass of `Item`. Platforms that support the AWT (such as some CDC platforms) return an instance of an object that subclasses `Component`.

■ **Note** On MIDP devices, if you want to render the video full-screen, you can pass the `VideoControl.USE_DIRECT_VIDEO` value to `initDisplayMode` to instruct it to draw the video directly to a `Canvas` object. Pass a reference to the desired destination `Canvas` object as the second argument. Set the `Canvas` you passed to `initDisplayMode` as the current `Displayable` item using `Display.getDisplay.setCurrent`, place the `Player` instance in the started state, and voilà!

As I note in the previous section, `Player` objects are best used in their own thread. Much of what they do is asynchronous and typically spans the entire stack of hardware and software on a device. It's important for you to have a way to get information about what the `Player` instance is doing. A simple way to do this is to use its `getState` method, which returns the current state that the `Player` instance is in. This is fine for coarse-grained control, especially if you manage an instance from different threads, but to really see what's going on, you can add a listener to the `Player` instance using its `addPlayerListener` method. The object you pass to `addPlayerListener` must implement the `PlayerListener` interface, which specifies the `playerUpdate` method that the `Player` instance invokes with information about media rendering. The event system defined by the MMAPI is extensible; the `PlayerListener` interface defines a number of events, which are shown in Table 16-4. The `PlayerListener` interface also lets individual `Player` objects pass proprietary events as values in a `String`. Your application might use some of these to provide fine-grained updates to its user interface—for example, to indicate that the device is buffering a stream.

Table 16-4. *Events Provided by Player Objects*

Event	Purpose
`PlayerListener.BUFFERING_STARTED`	The `Player` instance enters a buffering mode.
`PlayerListener.BUFFERING_STOPPED`	The `Player` instance leaves the buffering mode.
`PlayerListener.CLOSED`	The `Player` instance is closed.
`PlayerListener.DEVICE_AVAILABLE`	The exclusive device seized by the system has been returned to the `Player` instance.

Event	Purpose
PlayerListener.DEVICE_UNAVAILABLE	The system has seized an exclusive device previously used by the Player instance.
PlayerListener.DURATION_UPDATED	The duration of the media rendered by the Player instance has been updated.
PlayerListener.ERROR	An error occurred processing the media.
PlayerListener.RECORD_ERROR	An error occurred recording the media.
PlayerListener.RECORD_STARTED	Recording has started.
PlayerListener.RECORD_STOPPED	Recording has stopped.
PlayerListener.SIZE_CHANGED	The size of the video being rendered has changed.
PlayerListener.STARTED	The Player instance has entered the started state.
PlayerListener.STOPPED	The Player instance has stopped rendering media in response to a stop invocation.
PlayerListener.STOPPED_AT_TIME	The Player instance has stopped rendering media as a result of a StopTimeControl.
PlayerListener.VOLUME_CHANGED	The volume of an audio device has changed.

Capturing Media

The ability of a growing number of Java ME devices to capture audio and video is exciting, because it enables new kinds of applications, such as those that perform music recognition, as well as different social interactions over the network. The MMAPI makes it almost too easy to capture pictures from a built-in image sensor; you only need to open a Player to a device's video sensor and invoke a VideoControl's getSnapshot method. Listing 16-7 shows some pseudocode for this.

Listing 16-7. *Capturing a Snapshot from an Imaging Sensor*

```
Player player;
VideoControl vc;
Form viewer;

private void startCamera() {
    try {
        player = Manager.createPlayer("capture://video");
        vc = (VideoControl) player.getControl("VideoControl");
        player.realize();
        player.prefetch();
```

```
        if (vc != null) {
            Item vi = (Item) vc.initDisplayMode(
                VideoControl.USE_GUI_PRIMITIVE, null);
            viewer.append(vi);
        }
        Display.getDisplay(this).setCurrent(viewer);
        player.start();
    } catch (MediaException e) {}
      catch (IOException e) {}
}

private void stopCamera() {
    try {
        if (player!=null) {
            player.stop();
            player.close();
            player = null;
        }
    } catch (MediaException e) {}
}

private void capture() {
    if (vc != null) try {
        byte[] imageBytes = vc.getSnapshot(null);
        // do something with imageBytes...
    } catch (MediaException e) {}
}
```

Caution All of these methods should be invoked on their own thread; running them on the UI thread will likely block the UI thread for an unacceptably long time, causing application stalls. On some platforms, this may cause other problems, too, such as hanging the application, especially if the Java platform's security manager wants to prompt the user to approve the application's use of the `getSnapshot` method. To see how to do this in the context of an actual multithreaded application, read the upcoming "Putting the MMAPI and the SVGAPI to Work" section, especially the "Capturing Images" subsection.

By now, you can probably muddle through the purpose of `startCamera`. It gets a new `Player` instance that uses the device's imaging sensor as a video source, realizes and initializes the instance, and obtains a `VideoControl` instance to display on the screen as a camera viewfinder. The `stopCamera` method is simple, too; it just tears down the `Player` instance.

To capture the image, the `capture` method invokes the `VideoControl` instance's `getSnapshot` method. This method takes the name of the desired image format and encoding options and returns the image encoded in the desired image format as a `byte` array. By default, `getSnapshot` uses the system's default encoding, which is the first field in the value of the `video.snapshot.encoding` system property.

You can specify the encoding options as a URL-encoded string, with the following arguments:

- `encoding`: Indicates the desired encoding, such as JPEG, PNG, or GIF
- `width`: Indicates the desired width for the encoded image in pixels
- `height`: Indicates the desired height for the encoded image in pixels

For example, you might pass `encoding=jpeg&width=160&height=120` to `getSnapshot` to receive a stamp-sized image encoded as a JPEG.

`getSnapshot` can throw an exception; expect a `SecurityException` if the application does not have permission to take the snapshot, or a `MediaException` if the desired format isn't supported or if the `Player` doesn't support taking snapshots. You might also get an `IllegalStateException`, but only if you try to invoke `getSnapshot` on a `VideoControl` instance whose `initDisplayMode` hasn't been called.

Unfortunately, while the code itself is simple, in practice, this operation is fraught with difficulties. First are the obvious problems you might expect: your application might run on a device without an imaging sensor or without integration between the MMAPI and the imaging sensor. That's usually easy to find out; as you remember from Table 16-2, a call to `System.getProperty` passing the value `supports.video.capture` tells you if the target device can in fact perform this operation. However, you may discover other problems, including the following:

- *Different locators*: Some devices reserve the locator `capture://image` for capturing still images, and `capture://video` for capturing video using a `RecordControl` instance (I say more about that in a moment).
- *Rendering a preview of the camera viewfinder doesn't operate predictably*: On some devices, you can only obtain a camera viewfinder if you use the `initDisplayMode` to specify a GUI primitive using `VideoControl.USE_GUI_PRIMITIVE`. Other devices only work if you use `VideoControl.USE_DIRECT_VIDEO` and provide a `Canvas` object on which the MMAPI draws the view that the sensor provides.
- *Finding a common encoding scheme*: Don't build your application assuming that all Java ME devices will provide the same image format or resolution, or the format or resolution that you request. Some devices can encode in multiple formats, and some can't. Some can scale the encoded image, and some can't—and some will only scale your image if you provide *both* the width and height parameters.

- *Weird behavior*: Some devices are manufactured with an image sensor mounted upside down or rotated 90° in one direction or the other to meet specific industrial design requirements. Unfortunately, on devices like this, it's not uncommon to find out that the native camera software knows about the rotation and corrects for it, while the Java ME MMAPI implementation does not and captures the images rotated or upside down.

Camera support with the MMAPI can be maddening, so check a device's system properties with `System.getProperties`, work carefully, and document your results. When in doubt, check the manufacturer's developer support forums; unless it's a new device, odds are you're not the first to run into a specific problem. And don't despair: help is in on the way. JSR 234, the Advanced Multimedia Supplements (AMMS) API, defines a set of supplemental MMAPI interfaces, including several camera-specific `Control` subinterfaces. While not widely deployed on devices as I write this, AMMS API promises to standardize the behavior of the imaging subsystem, as well as provide a suite of new `Control` subinterfaces for imaging, including the following:

- `javax.microedition.amms.control.camera.CameraControl`: Use this to determine and change things like shutter feedback, camera rotation, exposure mode, and image resolution.

- `javax.microedition.amms.control.camera.ExposureControl`: Use this to determine and change exposure settings.

- `javax.microedition.amms.control.camera.FlashControl`: Use this to determine and set the flash mode for the camera flash.

- `javax.microedition.amms.control.camera.FocusControl`: Use this to determine and set the focus mode and specific focus settings for variable-focus imaging sensors.

- `javax.microedition.amms.control.ImageFormatControl`: Use this to set the desired image format.

- `javax.microedition.amms.control.camera.SnapshotControl`: Use this to control burst still-image captures, taking multiple images in one capture request.

- `javax.microedition.amms.control.camera.ZoomControl`: Use this to determine and set digital and optical zoom options for variable-zoom imaging sensors.

You can find out if a specific device has support for JSR 234 by invoking `System.getProperty`, passing the string `microedition.amms.version`. Devices supporting the AMMS API have stricter conformance requirements than those that support the MMAPI, so odds are good that there will be greater consistency of behavior between devices that provide the AMMS API.

■ **Tip** The AMMS API that JSR 234 defines provides a lot more than better control over imaging sensors. It also includes interfaces for advanced audio features such as 3D audio localization, reverberation, and equalizers; radio tuner control; effects for layering and grouping `Player` instances; and media postprocessing. If you're working on multimedia applications for high-end Java ME devices, be sure to see if the devices you target support this API.

The `getSnapshot` method obviously only works for capturing a single frame from a video data source. Your application may want to record other data sources, such as the audio stream from the microphone for a voice recorder or music-recognition application. To capture other kinds of media, you use the `RecordControl` interface, which writes a copy of the recorded media to an `OutputStream` instance. Listing 16-8 shows an example derived from the documentation for JSR 135.

Listing 16-8. *Pseudocode Demonstrating the RecordControl Interface*

```
try {
    // Create a DataSource that captures live audio.
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    Player p = Manager.createPlayer("capture://audio");
    p.realize();
    RecordControl rc = (RecordControl)p.getControl("RecordControl");
    rc.setRecordStream(baos);
    rc.startRecord();
    p.start();
    Thread.currentThread().sleep(5000);
    p.stop();
    rc.stopRecord();
    rc.commit();
    byte[] b =  baos.toByteArray();
} catch (IOException ioe) {}
  catch (MediaException me) {}
  catch (InterruptedException e) {}
```

This example creates a `Player` instance that captures the current audio for five seconds, and it gets an array of bytes to the encoded data that was captured. You can then use this content elsewhere. For example, you can use it as data that you write to a file, upload to a web server for processing, or send via the WMA as an MMS enclosure (which I discuss in Chapter 14).

When using a `RecordControl` instance, the MMAPI usually encodes the resulting data in the same format as the source. When recording from captured devices, you can find out the supported encoding scheme through the values of the `audio.encodings` and `video.encodings` system properties.

■ **Tip** As with the `getSnapshot` method, different devices may provide different encoding schemes, and media recorded on one device may not be playable on a different vendor's device. When designing your application, think carefully about the needs of different devices for specific media types, and keep in mind the possibility that your application will need a specific component (usually hosted on a server somewhere) to transcode media from one format to another.

Playing Individual Tones

One of my first experiences with writing a multimedia application wasn't on a computer, but rather on a programmable calculator. It had a `BEEP` instruction that let you play a four-tone beep encoded in ROM, and a `TONE` instruction that let you play a tone with a preassigned frequency. I thought this was the bee's knees at the time; being able to play a single tone is still useful for some applications.

The easiest way to play a tone or two is with the `Manager`'s `playTone` method, which takes a note, duration in milliseconds, and volume, like this:

```
Manager.playTone( 60, 3000, 100 );
```

The note value is actually a Musical Instrument Digital Interface (MIDI) note selector. Middle C is arbitrarily given the value `60`, and you count up or down by half steps from Middle C (e.g., A above Middle C would be `69`). As simple as it is, `playTone` can throw a `MediaException`, just like any player might, because another application could be using the sound hardware. It can also throw an `IllegalArgumentException` if the tone you provide is out of range.

If you want to play tones in sequence, it's better if you use a `Player` instance with an associated `ToneControl` instance, because then you can use the `Player` to control the sequencing of each tone. Follow these steps to play a sequence of tones:

1. Create a `Player` instance using the locator `Manager.TONE_DEVICE_LOCATOR` (which evaluates to `device://tone`).

2. Realize the player.

3. Get a `ToneControl` instance by invoking `player.getControl("ToneControl")`.

4. Pass a sequence of tones encoded as an array of bytes (`byte[]`) to the `ToneControl` instance you obtained in the previous step using the `ToneControl`'s `setSequence` method.

5. Start the `Player` instance.

If you think this is a little bizarre, rest assured you're not alone. Feeding a sequence of tones to a player via a control completely breaks the source-player-controller abstraction. Worse, the scheme to encode the tone sequence is a Byzantine affair reminiscent of note-control sequences defined by the MIDI standard, likely because most mobile devices have sound hardware that accepts MIDI commands. The byte stream that encodes the tone sequence consists of an initial version number indicating the version of the `ToneControl` instance you're using, followed by pairs of notes and durations. Unlike `playTone`, the duration you specify for the `ToneControl` instance isn't in milliseconds but is rather a multiple of the *resolution*—or fundamental tempo—of the `ToneControl` instance. By default, the resolution is $1/64^{th}$ of one measure of four beats (4/4 time, if you're a musician). Thus, in the default configuration, a duration of 64 is a whole note (four beats), a duration of 16 is a quarter note (one beat), a duration of 8 is an eighth note, and so on. Listing 16-9 shows an example of a simple sequence of notes for a `ToneControl` instance.

Listing 16-9. *Some Sample Data for a ToneControl Instance*

```
byte tempo = 30; // set tempo to 120 bpm
byte d = 8;      // eighth-note

byte C4 = ToneControl.C4;;
byte D4 = (byte)(C4 + 2); // two half steps, a whole step
byte E4 = (byte)(C4 + 4); // four half steps, a major third
byte G4 = (byte)(C4 + 7); // seven half steps, a fifth
byte rest = ToneControl.SILENCE;
byte block1 = 0;
byte block2 = 1;

byte[] song = new byte[] {
    // As transcribed in JSR-135
    ToneControl.VERSION, 1,
    ToneControl.TEMPO, tempo,
    ToneControl.SET_VOLUME, 50,
    ToneControl.BLOCK_START, block1,// Start defining a block
    E4,d, D4,d, C4,d, E4,d,         // content of block 1
    E4,d, E4,d, E4,d, rest,d,
    ToneControl.BLOCK_END, block1,  // end block definition
    ToneControl.BLOCK_START, block2,
    D4,d, D4,d, D4,d, rest,d,       // content of block 2
    E4,d, G4,d, G4,d, rest,d,
    ToneControl.BLOCK_END, block2,
    ToneControl.PLAY_BLOCK, block1, // play the first block
    ToneControl.PLAY_BLOCK, block2, // play the next block
    D4,d, D4,d, E4,d, D4,d, C4,d    // play the last section
};
```

This example shows a few other commands you can place in a sequence for a `ToneControl` instance. These commands influence the following options:

- *Tempo*: You can change the tempo of the playback at any time by using the `ToneControl.TEMPO` parameter, following it with the new tempo in beats per minute divided by `4`.
- *Volume*: You can change the volume at any time by using the `ToneControl.VOLUME` parameter, following it with the new volume (a number between `0` and `100`).
- *Blocks of notes*: You can define up to 255 blocks of notes to be repeated. To begin a block definition, use the `ToneControl.BLOCK_START` parameter, following it with a block ID. Follow this with the commands in the block, and close the block using the `ToneControl.BLOCK_START` parameter with the ID you used to start the block. You can then play the block at any time in the sequence by specifying the `ToneControl.PLAY_BLOCK` parameter and the block ID.
- *Silence*: Music isn't just about notes; it's also about the spaces between the notes. To specify a rest, use the `ToneControl.SILENCE` note value followed by the duration of the rest.
- *Repetition*: You can repeat a single note multiple times by specifying the `ToneControl.REPEAT` followed by the number of repetitions, then the note and duration. (For example, with the default resolution, the sequence `ToneControl.REPEAT, 4, 60, 16` plays four quarter notes on Middle C.)
- *Resolution*: You can change the resolution from the default of 1/64 using the `ToneControl.RESOLUTION` by following it with the denominator of the new resolution. However, you can do this only once at the beginning of a sequence.

Listing 16-10 builds on the pseudocode from Listings 16-1 and 16-9 to play "Mary Had a Little Lamb."

Listing 16-10. *Pseudocode to Play a Simple Song with a ToneControl Instance*

```
Player player;
ToneControl control;
byte tempo = 30; // set tempo to 120 bpm
byte d = 8;      // eighth-note

byte C4 = ToneControl.C4;;
byte D4 = (byte)(C4 + 2); // two half steps, a whole step
byte E4 = (byte)(C4 + 4); // four half steps, a major third
byte G4 = (byte)(C4 + 7); // seven half steps, a fifth
```

```
byte rest = ToneControl.SILENCE;
byte block1 = 0;
byte block2 = 1;

byte[] song = new byte[] {
    // As transcribed in JSR-135
    ToneControl.VERSION, 1,
    ToneControl.TEMPO, tempo,
    ToneControl.SET_VOLUME, 50,
    ToneControl.BLOCK_START, block1,// Start defining a block
    E4,d, D4,d, C4,d, E4,d,         // content of block 1
    E4,d, E4,d, E4,d, rest,d,
    ToneControl.BLOCK_END, block1,  // end block definition
    ToneControl.BLOCK_START, block2,
    D4,d, D4,d, D4,d, rest,d,       // content of block 2
    E4,d, G4,d, G4,d, rest,d,
    ToneControl.BLOCK_END, block2,
    ToneControl.PLAY_BLOCK, block1, // play the first block
    ToneControl.PLAY_BLOCK, block2, // play the next block
    D4,d, D4,d, E4,d, D4,d, C4,d    // play the last section
};
try {
    player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    player.realize();
    player.prefetch();
    control = (ToneControl)player.getControl("ToneControl");
    control.setSequence(song);
    player.start();
} catch (MediaException pe) {}
  catch (IOException ioe) {}
  catch (Exception e) {}
```

Specifying sequences of notes in your application like this is cumbersome and unwieldy, so I suggest that if you take this approach, generate the array of `ToneControl` sequences and store them as resources in your JAR file, which you can load using `getClass().getResourceAsStream()`.

Finally, a tip regarding whether to use the `playTone` method or a `ToneControl` instance: `playTone` does *not* block the currently executing thread. If you want to sequence multiple tones using `playTone`, you should do so in a separate thread and have the thread `sleep` between each invocation. Otherwise, one `playTone` may overwrite the next; at best, you'll hear silence, but you may just get garbled noise, too. For this reason, if you want to play multiple tones, I suggest you use a `Player` instance controlled by a `ToneControl` instance instead.

Introducing the Java Scalable 2D Vector Graphics API

These days, the idea of multimedia encompasses audio and video from analog sources, such as Hollywood movies, MP3 files of the Grateful Dead bootlegs, or photo snapshots from your camera phone. All-digital multimedia is booming, however, in the form of the SVG format. SVG is an open XML specification for describing two-dimensional vector graphics images that may be static or may include animation and support for user interaction. To compete with Adobe's Shockwave Flash (SWF) format, the mobile industry is widely adopting a subset of the SVG format, called SVG Tiny, for wireless terminals and other devices.

Understanding Basic SVG Concepts

The SVG format lets you represent an image using objects such as labels, circles, lines, polygons, and curves. Unlike the images to which you're probably accustomed, such as PNG, GIF, BMP, or JPEG images (which are called *bitmap* or *raster* images because they're composed of rows of individual picture elements), the SVG format is a *vector* format that uses a mathematical description of the shapes that make up an image. In addition, SVG images can contain bitmap data as well. This information is expressed using XML as defined by the W3C; for information on the specific standards, consult `http://www.w3c.org/TR`.

Because SVG is a vector format, images represented using SVG don't show scaling artifacts when rendered at different resolutions the way bitmap images do. Moreover, the standard explicitly provides support for animated images, letting artists create animations that render well on different-sized screens. Even more useful, SVG includes support for an event model that lets SVG images exchange events with the container that renders the image, so it's possible to express the look and feel of an entire user interface using SVG. SVG documents can also include scripts to handle events, making it a full-blown graphical authoring environment in its own right. Many browsers and other applications provide support for SVG already, making it a powerful standard with wide industry adoption.

JSR 287 defines an optional set of packages that support rendering and creating SVG images that comply with the SVG Mobile 1.2 standard (also known as SVG Tiny 1.2, described by the W3C at `http://www.w3.org/TR/2003/REC-SVGMobile-20030114/`). This standard eliminates two key features of SVG—support for scripts, and filters that the render can apply on a vector graphic to produce a modified bitmap output—as well as a few other limitations on clipping and transparency.

SVG is an XML application, so it's possible for you to create SVG content using nothing more than a text editor. Listing 16-11 shows an example.

Listing 16-11. *A Simple SVG Tiny Document*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg [
    <!ENTITY smile "
    <rect x='0' y='0' width='128' height='128' fill='gray' stroke='black'/>
    <g transform='translate(0, 0)'>
    <circle cx='64' cy='64' r='64' fill='yellow'/>
    <circle cx='40' cy='40' r='6.5' fill='black'/>
    <circle cx='88' cy='40' r='6.5' fill='black'/>
    <path d='M 40 88 L 64 96 88 88' stroke='black' stroke-width='2'/>
    </g>
    ">
    ]>
<svg xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
    <title>Smiley face</title>
    &smile;
</svg>
```

This code produces the image you see in Figure 16-3.

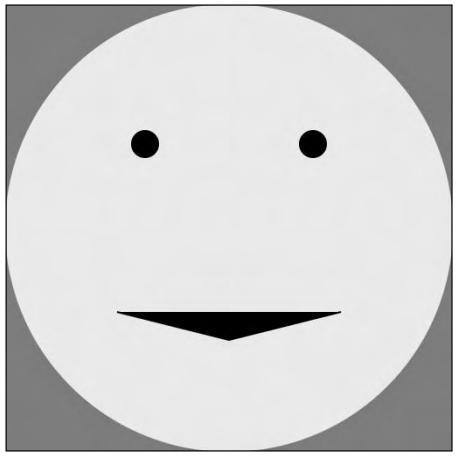


Figure 16-3. *A simple SVG image, rendered*

In practice, however, most SVG images of any use are too complex to code by hand; the vast number of objects in a complex image would simply take too long to produce manually. Fortunately, most vector graphics applications, from heavyweights like Adobe Illustrator to numerous smaller applications written by small shops and open source teams such as Inkscape (`http://www.inkscape.org/`), let you export images to SVG; these tools should be familiar to any artist who produces digital content.

A full exploration of the SVG and SVG Tiny formats is beyond the scope of this chapter; if you're interested in the nuts and bolts of the standard, you can start by looking at the W3C standards at `http://www.w3c.org/TR` and the wealth of information available on the Internet. In print, a good place to start is Kurt Cagle's discussion of the standard in his book *SVG Programming: The Graphical Web* (Apress, 2002).

Understanding the Organization of the SVGAPI

SVG support for Java ME began with JSR 226. Compatible with SVG Tiny 1.1, it provided a set of packages that includes support for basic rendering of SVG images and animated images, as well as handling user events from SVG images. The standards community quickly replaced JSR 226 with JSR 287, which provides better support for animation and an SVG document's DOM.

■ **Tip** As I write this, NetBeans 6.1 and the majority of devices with any SVG support only provide support for the SVGAPI that JSR 226 provides. Moving forward, however, I believe that the majority of SVG-enabled devices and development environments will support JSR 287, so throughout the remainder of this chapter, I focus on JSR 287.

The SVGAPI that both JSR 226 and JSR 287 describe provides support for several aspects of vector graphics rendering. Key, of course, is the high-level rendering API that JSR 287 provides. However, JSR 287 also provides interfaces that provide an encapsulation of the SVG DOM. It also provides SVG events, SVG animation through the Synchronized Multimedia Integration Language (SMIL), and an optional package that defines interfaces for high-performance immediate-mode rendering (in which the API does not keep a full model of the objects being rendered) that vendors can implement to give applications access to hardware rendering engines.

The SVGAPI provides the following packages:

- `javax.microedition.m2g`: The high-level interfaces for document rendering
- `javax.microedition.vectorgraphics`: Optional high-performance, immediate-mode rendering APIs that the SVGAPI and other applications can use
- `org.w3c.dom`: Interfaces that provide a subset of the DOM Level 3 Core API for managing XML DOMs
- `org.w3c.dom.events`: Event interfaces for the DOM Level 3 Core API
- `org.w3c.dom.smil`: Interfaces for rendering SMIL animation

- `org.w3c.dom.svg`: Interfaces and classes for the SVG DOM API
- `org.w3c.dom.views`: The DOM Views API that JSR 287 defines for interacting with the SVG DOM

Figure 16-4 shows a class diagram that illustrates the relationships between the relevant parts of the SVGAPI that I discuss in the following sections. In the figure, I omit package names for brevity.

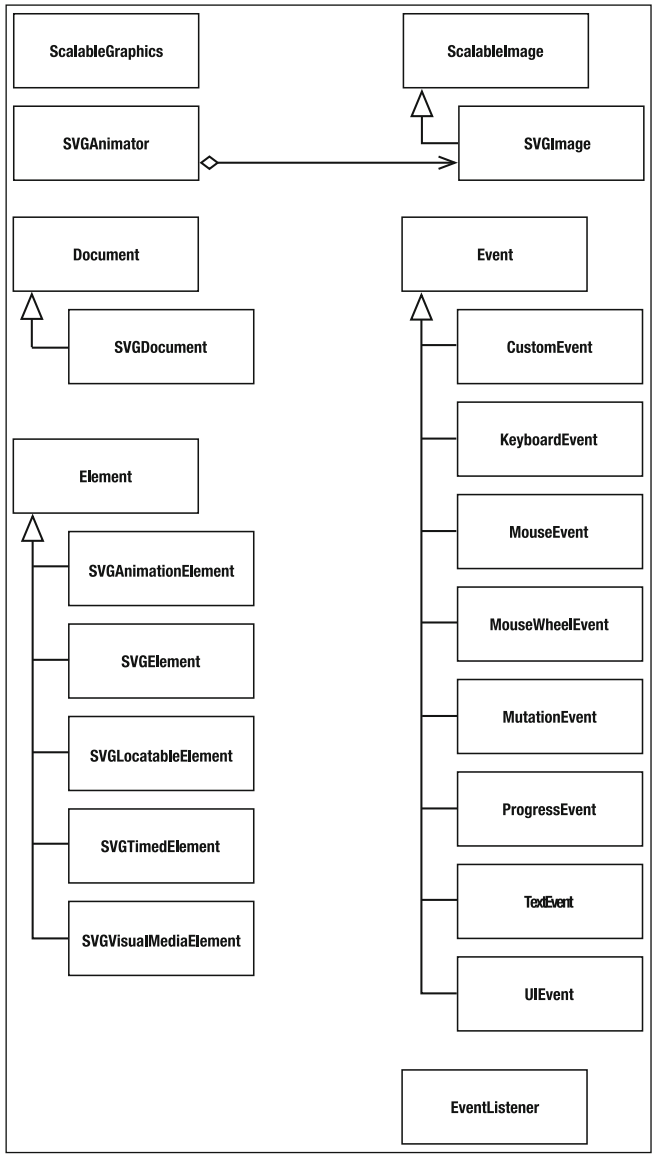


Figure 16-4. *The relationships between key parts of the SVGAPI*

■ **Note** The SVGAPI is big, especially if you take into account the APIs that the `javax.microedition.vectorgraphics` package provides. Figure 16-4 shows only the relationships between the classes you're likely to use when rendering SVG images.

As Figure 16-4 shows, the class and interface hierarchy is actually fairly flat for the root-level classes and interfaces. Be aware of the following classes and interfaces:

- `ScalableGraphics`: The fundamental class for 2D vector rendering
- `ScalableImage`: Represents an image in vector format; the parent of the `SVGImage` class
- `SVGImage`: Represents an SVG image conforming to SVG Tiny 1.2
- `SVGAnimator`: Handles automatic rendering of updates in animated SVG images using an `SVGImage`
- `Document`: Represents an XML document
- `SVGDocument`: Represents an SVG document; a child of `Document`
- `Element`: Represents a generic item in an SVG document's DOM; the parent to individual SVG elements
- `Event`: Represents contextual information about a specific interface; the parent of the SVG event classes
- `EventListener`: Represents a listener to events, specifying the `handleEvent` that the SVGAPI framework uses to send an object events

Rendering SVG Images

You can use the SVGAPI to render an SVG image in one of two ways. The first way is easy and best used for static images (those that do not require animation). The second is a little more complex, because it requires you to do everything you'd do for the first way and then a little more; however, in return, you get an object that manages an animated SVG image.

Rendering Static SVG Images

The first way is simple: create an `SVGImage` using some initial data to draw, and then draw it by giving it to a `ScalableGraphics` instance. Listing 16-12 shows pseudocode for this way to draw an SVG image.

Listing 16-12. *Rendering an SVG Image*

```
InputStream in = …
SVGImage image;
ScalableGraphics sg = ScalableGraphics.createInstance();
Graphics g;
int w, h;
try {
    image = (SVGImage)ScalableImage.createImage(in, null);
} catch( IOException e) {};

sg.bindTarget(g);
image.setViewportWidth(w);
image.setViewportHeight(h);
sg.render(0, 0, image);
sg.releaseTarget();
```

When you create an `SVGImage` instance, you can specify the source data as an instance of an `InputStream` subclass or as a URL. When you create an instance, you can also specify an external resource handler that the `SVGImage` uses to load subelements. Usually, you don't need to do this—you can simply pass `null`. However, you might want to do this when your image has components to be fetched from over the network, for example; in that case, you'd provide an implementation of the `ExternalResourceHandler` that wraps the `HttpConnection`.

Of course, Listing 16-12 doesn't translate well to the MIDP graphics environment. Listing 16-13 builds on Listing 16-12 and shows a simple `SVGImageItem` class that extends the MIDP `CustomItem` class that I first discuss in Chapter 5. Instances of the `SVGImageItem` class are suitable for addition to `Form` items in your MIDlet.

Listing 16-13. *The SVGImageItem Class for Rendering a Static SVG Image on a Form*

```
import java.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.m2g.*;

public class SVGImageItem extends CustomItem  {
    private ScalableGraphics sg;
    private SVGImage image;
    private int prefWidth, prefHeight;
```

```
    public SVGImageItem( String l, InputStream in, ExternalResourceHandler h ) {
        super(l);
        sg = ScalableGraphics.createInstance();
        if (in != null) {
            try {
                image = (SVGImage)ScalableImage.createImage(in, h);
            } catch( IOException e) { e.printStackTrace(); };
        } else {
            image = (SVGImage)SVGImage.createEmptyImage(h);
        }
        SVGAnimator animator = SVGAnimator.createAnimator(image);
        Canvas svgCanvas = (Canvas)animator.getTargetComponent();
        prefWidth = svgCanvas.getWidth();
        prefHeight = svgCanvas.getHeight();
        image.setViewportWidth(prefWidth);
        image.setViewportHeight(prefHeight);
    }

    protected int getPrefContentHeight(int w) {
        return minContentHeight();
    }

    protected int getPrefContentWidth(int h) {
        return minContentWidth();
    }

    protected int getMinContentHeight() {
        return prefHeight;
    }

    protected int getMinContentWidth() {
        return prefWidth;
    }

    public void paint(Graphics g, int w, int h) {
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, w, h);
        sg.bindTarget(g);
        image.setViewportWidth(w);
        image.setViewportHeight(h);
        sg.render(0, 0, image);
        sg.releaseTarget();
    }
}
```

This class must meet the minimal requirements of the `CustomItem` interface by implementing a constructor that calls the inherited constructor, providing layout hints to the layout manager via the `getPref` and `getMin` dimension methods, and providing a `paint` method that draws the contents of the item.

The `SVGImageItem` constructor method in Listing 16-13 is a little more involved than the pseudocode in Listing 16-12, because it uses an `SVGAnimator` to obtain preferred bounds for the SVG image. I create an `SVGAnimator`, which would render on a `Canvas`, and then get the preferred bounds of that `Canvas` instance. After I do this, I just discard the `Canvas` instance and `SVGAnimator`, because I won't need it for anything else. Finally, I set the image's *viewport*—its bounds—to the preferred bounds I obtained from the `SVGAnimator`'s `Canvas` instance.

■ **Note** While it's possible to determine an SVG image's bounds by using the `SVGDocument` methods to access the DOM of the SVG image directly, using an `SVGAnimator` is clearer and will pave the way for you to understand how to animate an SVG image, which I show you in the next section.

The MIDP layout manager uses the `getPrefContentHeight` and `getPrefContentWidth` methods to determine the SVG image's preferred height and width, respectively, as computed by the constructor. When the layout manager wants to obtain a preferred bound for width or height, it passes the appropriate method a desired bound for the *other* axis. The `getMinContentHeight` and `getMinContentWidth` methods return the preferred bounds for the image.

The `paint` method performs the same basic work as the pseudocode in Listing 16-12. First, it paints an empty rectangle where it will paint the SVG image, in case something was on the screen at that location. Next, it binds the `ScalableGraphics` instance to the `Graphics` instance that the containing `Canvas` provided to the `paint` method. This provides the `ScalableGraphics` instance with a `GraphicContext` to use when rendering the SVG image. After that, it sets the image's actual width and height to the width and height provided by the containing `Canvas`, and then it tells the `ScalableGraphics` instance to render the image at the origin of the item. Finally, it releases the `Graphics` context bound using the `ScalableGraphics`'s `releaseTarget` method.

Rendering Animated SVG Images

The second way to render an `SVGImage`, using an `SVGCanvas`, is a little more work, but it lets you easily render animated SVG images or those that process events. The strategy is similar to what you saw in Listing 16-13:

1. Create an instance of `SVGImage` representing the image you want to draw.

2. Create an instance of `SVGAnimator` with the `SVGImage`.

3. Get the component—a `Canvas` when using the MIDP—on which the `SVGAnimator` will render the image.

4. Set the bounds of the `SVGImage` to the bounds the `SVGAnimator`'s component.

5. Make the `SVGAnimator`'s component visible.

6. Use the `SVGAnimator` to start playing the animation.

When rendering an animated SVG, the `SVGAnimator` implementation uses the desired SVG image and a container of some kind provided by the platform's actual GUI hierarchy, such as the MIDP or the AWT. The designers of the `SVGAnimator` class assumed that it would be used on both CLDC and CDC platforms, so it doesn't return a specific instance of an MIDP class such as `Canvas`. Instead, the `SVGAnimator` provides the `getTargetComponent` to return a component into which it renders the SVG, much as an MMAPI `VideoControl` instance returns a component via its `initDisplayComponent` method. When using the SVGAPI on an MIDP platform, the object that `getTargetComponent` returns is a `Canvas` instance; on other platforms, it's a subclass of `Component`. Thus, on MIDP platforms, you can use the `Display` class's `setCurrent` method to set the resulting `Canvas` as the `Displayable` to draw; on other platforms, you can insert the resulting component into the appropriate container in your GUI's view hierarchy.

The `SVGAnimator` class implements an update thread, which it uses to render the SVG image as an animated image. The thread itself implements a simple state machine with the three states that you see in Figure 16-5.

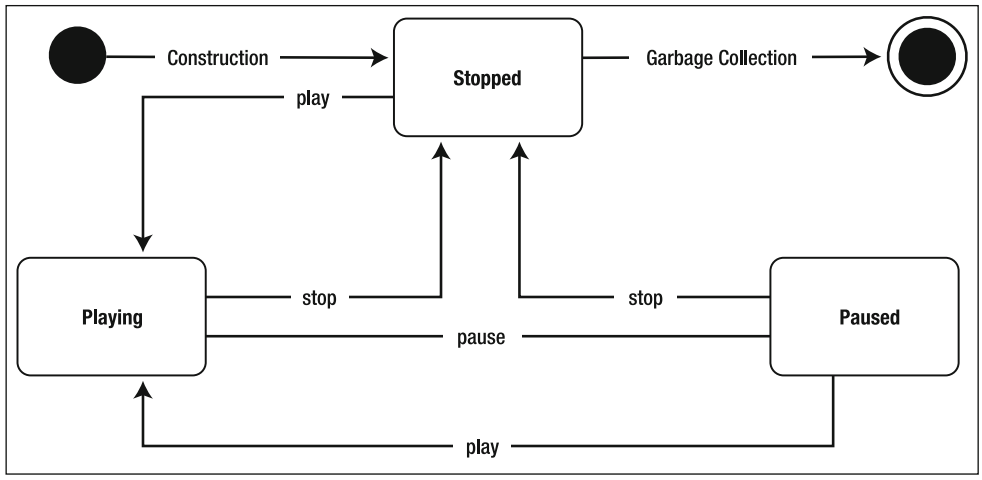


Figure 16-5. *The SVGAnimator state machine*

An SVGAnimator instance can be in one of the following states:

- *Stopped*: An SVGAnimator begins its life in the stopped state, and it does not draw anything during this state. When it's in the other states, you can place an SVGAnimator instance in the stopped state by invoking its stop method.
- *Playing*: An SVGAnimator with a running update thread that is drawing to its component is in the playing state. You can pause a playing SVGAnimator by invoking its pause method, or stop it by invoking its stop method.
- *Paused*: An SVGAnimator may be paused, during which time the animation is not updating and not proceeding. You can resume playing an SVGAnimator by invoking its play method while it's in the paused state, or stop it by invoking its stop method.

Listing 16-14 shows pseudocode that sets up an SVGAnimator and starts playback.

Listing 16-14. *Playing an Animated SVG*

```
SVGAnimator svgAnimator = null;
String mediaName = …;
Display display = …;
private void initAndStartSvgPlayer()
    throws IOException {
    InputStream in = getClass().getResourceAsStream(mediaName);
    SVGImage svgImage = (SVGImage)ScalableImage.createImage(in, null);
    svgAnimator = SVGAnimator.createAnimator(svgImage);
    Canvas svgCanvas = (Canvas)svgAnimator.getTargetComponent();
    svgImage.setViewportWidth(svgCanvas.getWidth());
    svgImage.setViewportHeight(svgCanvas.getHeight());
    display.setCurrent((Displayable)svgCanvas);
    svgAnimator.play();
}
```

Because the SVGAnimator uses a separate thread to perform the animation, you can't access the SVG image itself directly while the image is animating. Instead, if you need to change the contents of the image (see the next section, "Modifying SVG Images"), you should do so only using the SVGAnimator's invokeAndWait or invokeLater method.

■ **Note** Rendering an SVG image does *not* use the same API and concepts that the MMAPI defines. This is a shame, as a little work to extend the notions of data source and player would likely have made for a clean interface. When working with media, don't confuse the SVGAPI with the MMAPI.

Through its target object (the `Canvas` or `Component` subclass) as well as the SVG content itself, an `SVGAnimator` can obtain and generate events. For your application logic to receive these events, it must implement a class that inherits the `SVGEventListener` interface, which specifies methods for user events such as key-press events, key-release events, pointer-press events, pointer-release events, and notifications when the platform hides or shows the animation itself. You can process these events by hand, but most of the time when you want to interact with events from an SVG image, it will be to create an SVG-specific control. Fortunately, the NetBeans environment provides some classes that do this for you, as I discuss later in the "Using NetBeans with SVG Images" section.

■ **Note** JSR 226 defines a mechanism by which you can use the MMAPI to play an SVG image. It's essentially the same as playing other video; get a `Player` instance with a locator that specifies an SVG image, and then get a `Control` subinterface `javax.microedition.media.control.SVGControl`. Using that `Control` instance, you can get an `Item` or `Component` object to add to your view hierarchy and play the media normally. JSR 287 does not specify whether devices will continue to support the MMAPI for SVG playback.

Modifying SVG Images

While it's less likely that you will want to write an application that creates SVG images instead of displaying them, it's certainly possible, and the SVGAPI that JSR 287 defines lets you do it. Because the SVGAPI provides support for the SVG DOM, it's entirely possible for you to create an empty SVG image and add new objects to it. To do this, you must have working knowledge of the XML DOM, the SVG DOM, and the SVG standard itself; all of that is largely outside the scope of this section. Here, I just sketch the general principle so you know that it's something you can do if you find you have the need. All of what I say in this section is only on JSR 287–enabled devices; JSR 226 does not have many of the classes and some of the methods required if you want to work directly with the SVG DOM.

Listing 16-15 shows pseudocode derived from JSR 287 that gives an example of modifying an existing SVG image using an `SVGAnimator` to provide user events to application logic. In turn, those user events cause the code to add new circles to the SVG image that the application is displaying.

Listing 16-15. *Creating and Populating an SVG Image*

```
class MIDPSVGEditor implements SVGEventListener, Runnable {
    private static final String svgNS = "http://www.w3.org/2000/svg";
    private SVGDocument svgDocument;
    private SVGAnimator svgAnimator;
    private Vector addToTree = new Vector();
    private int cx, cy;

    public MIDPSVGEditor (SVGImage svgi) {
        svgDocument = svgi.getDocument();
        svgAnimator = SVGAnimator.createAnimator(svgi);
        svgAnimator.setSVGEventListener(this);
    }

    public Object getTargetComponent() {
        return svgAnimator.getTargetComponent();
    }

    public void pointerPressed(int x, int y) {
    }

    public void pointerReleased(int x, int y) {
        cx = x;
        cy = y;
    }

    public void keyPressed(int keyCode) {
        SVGElement circle = svgDocument.createElementNS(svgNS, "circle");
        circle.setFloatTrait("cx", (float)cx);
        circle.setFloatTrait("cy", (float)cy);

        synchronized (addToTree) {
           addToTree.addElement(circle);
        }

        svgAnimator.invokeLater(this);
    }

    public void keyReleased(int keyCode) {
    }
```

```
    public void run() {
        synchronized (addToTree) {
          for (int i = 0; i < addToTree.size(); i++) {
            svgDocument.getDocumentElement().appendChild(
               (SVGElement) addToTree.elementAt(i));
          }
          addToTree.removeAllElements();
        }
    }

    public void hideNotify() {
    }

    public void showNotify() {
    }

}
```

■ **Note** Listing 16-15 *won't* work with JSR 226, as JSR 226 does not include the `SVGDocument` class, support for the SVG DOM, or the ability to create empty SVG images with interaction hints. While it's possible to create SVG XML on a Java ME device and render it using JSR 226, you won't have the flexibility that the `SVGDocument` and other JSR 287 classes provide that let you work with individual SVG DOM elements.

You initialize this call with a preexisting `SVGImage`. The image might be one from another source, or you can create an empty `SVGImage` by invoking the `SVGImage` `static` method `createEmptyImage`, like this:

```
SVGImage svgImage = SVGImage.createEmptyImage( null, hints );
```

The first argument to `createEmptyImage` is an optional `ExternalResourceHandler`, just as if you were creating an `SVGImage` for an existing image. The second argument is information about the kind of interactions the `SVGImage` instance should expect from users and the SVG content itself. It's a bit mask that can contain the following values, binary ORed:

- `SVGImage.INTERACTION_MODE_DISABLED`: Indicates that no user interaction with the SVG image will take place
- `SVGImage.INTERACTION_MODE_SCRIPT_EXECUTION`: Indicates that script execution within the SVG image may take place (with most implementations of the API, this has no effect)

- `SVGImage.INTERACTION_MODE_FOCUS_HIGHLIGHT`: Indicates that the SVG image may contain focus highlighting, and the user may change which element has focus
- `SVGImage.INTERACTION_MODE_IN_PLACE_TEXT_EDIT`: Indicates that the SVG image may contain in-place text editing (within a text field, for example)
- `SVGImage.INTERACTION_MODE_OTHER`: Indicates that the implementation should support other user interaction with the SVG image

In Listing 16-15, the `keyPressed` and `run` methods show you how to mutate an SVG image. When you press a key while the `SVGAnimator`'s target component has focus, the `SVGAnimator` invokes `keyPressed`, which creates a new circle element that it will add to the target document later at the first thread-safe opportunity. Because the `SVGAnimator` uses multiple threads, the code needs to synchronize the addition of the circle element with the `SVGAnimator`; the code adds the new circle to a vector and schedules the update using `invokeLater`. The `run` method—which the Java VM invokes at the appropriate time—actually adds any pending circles to the SVG document.

Generally speaking, the `Document` and `SVGDocument` classes let you interact with a specific SVG document; this example uses the `Document` method `createElementNS` to create an individual circle to draw on the image. To use `createElementNS` correctly, you need to have a pretty good grip of the SVG standard, because you use it to create primitives in the SVG language. If you're going this route in your application, I suggest you carefully study the W3C standards I cited previously.

Of course, with these classes you can also query a specific document's DOM; the `run` method in Listing 16-15 does this to get the document element of the SVG XML, which by the definition of SVG has a child for each shape in the image. You can change elements of the DOM, too, using the `Node` interface that the `SVGDocument` inherits; `run` does this by invoking `appendChild`, passing the new object `createElementNS` created.

Using NetBeans with SVG Images

The approach that the SVGAPI takes to providing SVG support is fine for many applications, but it's a little labor intensive, especially for things many application developers would like to do, such as simply play an SVG animated image, show an SVG image as a splash screen or wait screen, or use one as a menu. While you can certainly write code to do this yourself—it's a matter of composing a `Displayable` that uses an `SVGAnimator`—it's going to be code that you, your virtual neighbor, and a good deal many other Java ME developers are going to be writing over and over again for the next few years. Fortunately, there's a better way.

The Mobility Pack for the NetBeans IDE provides a small package, `org.netbeans.microedition.svg`, which contains some utility classes for building SVG user interfaces. These classes all extend the MIDP `Canvas` class, so you simply use the `Display` class's `setCurrent` to display an instance of one. Even better, the classes are integrated with the

NetBeans IDE, so if you're building your application using the visual designer, you can drag out the components you want to use in your application, just as you would for conventional MIDP components, as you saw in Chapter 3. The `org.netbeans.microedition.svg` class includes the following classes:

- `SVGPlayer`: Provides a simple wrapper around the SVGAPI to play animated SVG images. It accepts the events it receives as a `Canvas` implementation and forwards them to the SVGAPI.
- `SVGSplashScreen`: Provides a screen for showing a single SVG animation that transitions to another screen.
- `SVGWaitScreen`: Provides an interstitial screen that displays an SVG animation while a blocking background task executes.
- `SVGMenu`: Provides a player for an SVG that contains multiple selectable items. Menu items are individual SVG elements in a single SVG image.

Note These classes are provided in the version of the `org.netbeans.microedition.svg` package that NetBeans 6.1 includes. Previous versions of NetBeans had similar classes, although in those releases, the `SVGPlayer` didn't exist; instead, it included the `SVGAnimatorWrapper`. If you're working with older versions of NetBeans, you'll want to either upgrade or use `SVGAnimatorWrapper`. Upgrading is probably the right choice to make, because `SVGAnimatorWrapper` is deprecated in more recent releases of the package.

It's easy to use the NetBeans visual GUI builder to create an application that uses these classes, but of course you don't have to; you can also import the `org.netbeans.microedition.svg` package and use these classes directly.

These classes and others are under constant refinement by the NetBeans developer community; if you're interested in working with SVG content at a higher level than the SVGAPI, you should consult the latest documentation at `http://wiki.netbeans.org/`.

Putting the MMAPI and the SVGAPI to Work

You could build some interesting weather applications around the MMAPI and the SVGAPI, such as a server-side application that provides weather reports as SVG animations to Java ME clients, or a weather-blogging application that permits users to view, share, and interact with weather data. However, they'd all be pretty complex. Consequently, rather than integrating the MMAPI or the SVGAPI with the WeatherWidget example I've touched upon throughout this book, in this section I present a simple media player application that plays media using both the MMAPI and the SVGAPI. The application opens with a list of media

files from which you may choose one to play. Figure 16-6 shows the application's main screen—a media player—displaying a single frame of a movie.

Figure 16-6. *The media player view for the MultimediaMIDlet application showing a frame from* Duck and Cover[2]

Note As I alluded to previously in the "Capturing Media" section, support for media capture on Java ME devices is a hit-or-miss affair, and the simulator is no exception. Worse, the simulation environment that the Mobility Pack for NetBeans provides doesn't support the full range of media files that today's average Java ME device can render. When debugging media applications, I find it best to test using WAV audio files or MPEG-1 video files in the simulator, and then move to a target handset on which I know what audio and video formats are actually available. Multimedia support on today's Java ME devices is a rapidly changing part of the Java ME ecosphere, so be prepared to spend some time researching supported media types for specific devices and testing when building your application.

2. *Duck and Cover*, directed by Anthony Rizzo (1951; Federal Civil Defense Administration, Archer Productions). Now in the public domain; see `http://en.wikipedia.org/wiki/Duck_and_Cover_(film)`, `http://www.imdb.com/title/tt0213381/`, and `http://www.archive.org/details/DuckandC1951`.

Listing 16-16 shows the entire `MultimediaMIDlet` class, which provides the entire implementation. As a MIDlet, its target is of course the MIDP; you could write similar code using the MMAPI and the SVGAPI to run on a CDC platform that included these APIs in addition to AWT or Swing.

Listing 16-16. *The MultimediaMIDlet Class*

```
package com.apress.rischpater.multimediamidlet;

import java.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.m2g.*;

public class MultimediaMIDlet
    extends MIDlet
    implements CommandListener, Runnable {
    private Display display;
    private List choiceScreen;
    private Form mediaScreen;
    Displayable viewerDisplayable;
    private Item videoItem;
    private ImageItem capturedImage;
    private VideoControl videoControl;
    private Command captureCommand;
    private Command selectCommand;
    String mediaName = null;
    String mediaType = null;
    private Player player = null;
    SVGAnimator svgAnimator;
    boolean endNow;
    private static final String SVG_IMAGE_PATH = "/res/image.svg";
    private static final String SVG_IMAGE_TYPE = "image/svg+xml";
    private static final String WAV_SOUND_PATH = "/res/sound.wav";
    private static final String WAV_SOUND_TYPE = "audio/x-wav";
    private static final String MPG_MOVIE_PATH = "/res/movie.mpg";
    private static final String MPG_MOVIE_TYPE = "video/mpeg";
    private static final String CAP_VIDEO_PATH = "capture://video";
    private static final String CAP_VIDEO_TYPE = "video/x-capture";
```

```
private void init() {
    showSupportedMedia();
    display = Display.getDisplay(this);
    if (choiceScreen == null) {
        choiceScreen = new List("MultimediaMIDlet", List.IMPLICIT);
        choiceScreen.addCommand(new Command("Exit", Command.EXIT, 0));
        selectCommand = new Command("Play", Command.ITEM,1);
        choiceScreen.setSelectCommand(selectCommand);
        choiceScreen.addCommand(selectCommand);
        choiceScreen.setCommandListener(this);
        choiceScreen.append("Sound", null);
        choiceScreen.append("Video", null);
        choiceScreen.append("SVG", null);
        choiceScreen.append("Camera Capture",null);
    }
    display.setCurrent(choiceScreen);
}

public void startApp() {
    init();
}

private void showSupportedMedia() {
    String[] contentTypes = Manager.getSupportedContentTypes(null);
    for (int i=0; i<contentTypes.length; i++) {
        String protocols[] =
            Manager.getSupportedProtocols(contentTypes[i]);
        for (int j=0; j<protocols.length; j++) {
            String s = contentTypes[i] + ":" + protocols[j];
            System.out.println(s);
        }
    }
}

public void pauseApp() {
    endNow = true;
    if (svgAnimator != null) {
        svgAnimator.stop();
    }
    if (player!=null) {
        player.close();
    }
}
```

```
    public void destroyApp(boolean unconditional) {
        endNow = true;
        if (player != null)  {
                player.close();
        }
    }

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT) {
            destroyApp(true);
            notifyDestroyed();
        } else
        if (display.getCurrent() == viewerDisplayable) {
            if (c == captureCommand) {
                synchronized(this) {
                    this.notify();
                }
            }
        } else
        if (display.getCurrent() == choiceScreen ) {
            if (c == selectCommand ) {
                Form waitForm = new Form("Loading...");
                display.setCurrent(waitForm);
                int itemIndex = choiceScreen.getSelectedIndex();
                switch(itemIndex) {
                    case 0:
                        mediaName = WAV_SOUND_PATH;
                        mediaType = WAV_SOUND_TYPE;
                        break;
                    case 1:
                        mediaName = MPG_MOVIE_PATH;
                        mediaType = MPG_MOVIE_TYPE;
                        break;
                    case 2:
                        mediaName = SVG_IMAGE_PATH;
                        mediaType = SVG_IMAGE_TYPE;
                        break;
                    case 3:
                        mediaName = CAP_VIDEO_PATH;
                        mediaType = CAP_VIDEO_TYPE;
                        break;
                }
```

```
            Thread t = new Thread(this);
            t.start();
        }
    }
}

public void run() {
    playFromResource();
    while(!endNow) {
        synchronized(this) {
            try {
                this.wait();
            } catch (Exception e) {}
            if (!endNow) {
                try {
                    byte[] raw = videoControl.getSnapshot(null);
                    Image image = Image.createImage(raw, 0, raw.length);
                    capturedImage.setImage(image);
                } catch (MediaException e) {continue;}
            }
        }
    }
}

private void initMediaPlayer()
    throws IOException, MediaException {
    if (mediaScreen == null) {
        mediaScreen = new Form("MultimediaMIDlet");
        mediaScreen.addCommand(new Command("Exit", Command.EXIT, 0));
        mediaScreen.setCommandListener(this);
    }
    viewerDisplayable = (Displayable)mediaScreen;
    if (mediaType.equals(CAP_VIDEO_TYPE))
        player = Manager.createPlayer(mediaName);
    else {
        InputStream in = getClass().getResourceAsStream(mediaName);
        player = Manager.createPlayer(in, mediaType);
    }
    player.realize();
    player.prefetch();
}
```

```
    private void initSvgPlayer()
        throws IOException {
        InputStream in = getClass().getResourceAsStream(mediaName);
        SVGImage svgImage = (SVGImage)ScalableImage.createImage(in, null);
        svgAnimator = SVGAnimator.createAnimator(svgImage);
        Canvas svgCanvas = (Canvas)svgAnimator.getTargetComponent();
        viewerDisplayable = (Displayable)svgCanvas;
        svgImage.setViewportWidth(svgCanvas.getWidth());
        svgImage.setViewportHeight(svgCanvas.getHeight());
}

    private void configViewSound() {
        // No-op
    }

    private void configViewSvg() {
        // No-op
    }

    private void configViewVideo() {
        captureCommand = new Command("Capture", Command.ITEM, 0);
        mediaScreen.addCommand(captureCommand);
        videoControl = (VideoControl) player.getControl("VideoControl");
        if (videoControl != null) {
            videoItem = (Item) videoControl.initDisplayMode(
                    VideoControl.USE_GUI_PRIMITIVE, null);
            mediaScreen.append(videoItem);
        }
        capturedImage = new ImageItem(null, null,
            ImageItem.LAYOUT_DEFAULT, null);
        mediaScreen.append(capturedImage);
    }

    private void playFromResource() {
        try {
            if ( mediaType.equals("image/svg+xml")) {
                initSvgPlayer();
                configViewSvg();
                svgAnimator.play();
            }
```

```
            else
            {
                initMediaPlayer();
                if (mediaType.startsWith("audio")) {
                    configViewSound();
                } else if (mediaType.startsWith("video")) {
                    configViewVideo();
                }
                player.start();
            }
            display.setCurrent(viewerDisplayable);
        }  catch (Exception e) {
            showException(e);
        }
    }

    private void showException(Exception e) {
        Alert a = new Alert("Exception", e.toString(), null, null);
        a.setTimeout(Alert.FOREVER);
        display.setCurrent(a, viewerDisplayable);
   }
}
```

The MIDlet source opens with a few import directives, because it relies on the MIDP APIs as well as the MMAPI and the SVGAPI. The MIDlet follows the basic organization of all MIDlets in the book, eschewing initialization in the constructor in favor of an explicit `init` method that the `startApp` method invokes when the AMS launches the MIDlet. The following methods are responsible for MIDlet behavior:

- `startApp`, `pauseApp`, *and* `destroyApp`: These methods together manage the life cycle of the MIDlet. Starting the MIDlet shows a list of media to play, while pausing or quitting the MIDlet stops any pending media playback.

- `init`: As previously mentioned, `init` creates the initial `List` instance showing the choices for media playback. It also dumps a list of supported media types to the system console using the `showSupportedMedia` method.

- `commandAction`: This method processes user selections from the initial `List` screen and permits you to capture a video frame during video playback.

- `run`: The application uses a separate thread (which I'll call the *player thread*) for media playback; this thread executes the `run` method, which starts media playback and waits for a request from the user to capture media.

- `initMediaPlayer` *and* `initSvgPlayer`: These methods initialize the classes in the MMAPI and the SVGAPI, respectively.

- `configViewSound`, `configViewSvg`, *and* `configViewVideo`: These methods perform additional configuration of the `Canvas` subclass instance that displays the media. Some are no-ops and don't do anything in this implementation; they're provided for illustration only.

- `playFromResource`: This method uses the media type you select to determine whether to initialize the MMAPI or the SVGAPI, start the media playback, and show the `Canvas` subclass instance that contains the rendered media. The player thread's `run` method invokes this method.

- `showException`: This method shows an exception in an `Alert`.

The `showSupportedMedia` method is one of those utilities you never think you have to write, yet you find yourself writing it again and again. It queries the MMAPI `Manager` for a list of supported content types using its `getSupportedContentTypes` function. With the content type (e.g., `audio/wav`), it queries the `Manager` again to determine which protocols the implementation of the MMAPI supports (e.g., HTTP). Finally, it prints each supported pair to the console. Listing 16-17 shows representative output from the simulator.

Listing 16-17. *Supported Media Content Types and Protocols*

```
video/mpeg:http
video/mpeg:file
image/gif:http
image/gif:file
audio/x-wav:http
audio/x-wav:file
audio/x-wav:capture
audio/amr:http
audio/amr:file
audio/x-tone-seq:http
audio/x-tone-seq:file
audio/x-tone-seq:device
video/vnd.sun.rgb565:capture
audio/sp-midi:http
audio/sp-midi:file
audio/midi:http
audio/midi:file
audio/midi:device
```

You might think you don't need this information—after all, the vendor information that accompanies most Java ME devices provides a list of supported media types—but frequently you do, because there's no guarantee that a Java ME device's firmware is wired correctly with the MMAPI just because it can render a media type using a native application. In fact, it's possible at times that the media list that the MMAPI Manager provides may be incorrect.

■ **Tip** While testing *any* Java ME application on a device is a crucial part of your development cycle, it's especially so with most of the Java ME extensions documented through JSRs, and the MMAPI is no different. Always test your application on target hardware early in your development cycle, and repeat the testing often.

The heart of the UI flow is the `commandListener` method and the MIDlet fields `mediaName` and `mediaType`. When you choose a kind of media to play, `commandListener` uses a simple `switch` statement to populate these variables with the file name in the JAR file and the type of media you selected. Next, it launches the player thread, using the MIDlet class itself as the `Runnable` item. The resulting thread starts with the MIDlet class's `run` method, which initializes and starts playback.

The next sections describe how the MIDlet plays audio and video, captures images from the video stream, and plays animated SVG content.

Playing Audio and Video

When the player thread starts, its `run` method first invokes `playFromResource`. This method uses the information the `commandListener` set aside about the media type you selected. Later, the implementation uses this information to determine whether to initialize the SVGAPI or the MMAPI, as well as to set the next `Displayable` item in the `viewerDisplayable` field of the MIDlet. In addition, if you selected a file that isn't an SVGAPI, the class will create any additional controls the MIDlet requires. This logic consists of a series of nested `if-else` statements, which are sufficient for this example, but possibly worth dividing into separate subclasses with a common interface that returns a `Displayable`.

The `initMediaPlayer` does the heavy lifting initializing the MMAPI; it closely resembles Listing 16-1. It creates a `Form` instance, which becomes the `Displayable` that the MIDlet shows when rendering the media. `initMediaPlayer` adds an instance of `Command` that permits you to exit the application before setting the command listener for the new form to the MIDlet.

The method then creates a `Player` instance, using either the locator of the camera if you selected the Camera Capture option, or an `InputStream` taken from the JAR resource named by your choice in the `commandAction` method. Finally, it configures the new `Player` instance by first realizing the player and then prefetching the content.

Once `initMediaPlayer` does its work, control returns to `playFromResource`, which invokes either `configViewSound` or `configViewVideo` to do any additional postconfiguration of the user interface before starting the media playback. The `configViewSound` method does nothing; it's simply a placeholder. You might want to experiment with the code and create various `Control` subclasses on the `player` field here, such as a `VolumeControl`. `configViewVideo`, however, must do a little work; it creates another `Command` instance that lets you trigger a video capture and adds it to the `Form` instance before creating the `VideoControl` instance. The resulting `VideoControl` provides the method with an `Item` instance to add to the `Form`. Finally, the method creates a second additional empty `ImageItem` instance that the MIDlet uses to display the image you capture when you select the Capture option.

The `playFromResource` method then starts the newly created `Player` instance and sets the current `Viewable` instance to the `viewerDisplayable` before returning control to the player thread's `run` method. At this point in the MIDlet execution, the MMAPI renders the media you selected, while the player thread blocks on `this` waiting for you to select the Capture option.

Capturing Images

The `VideoControl` instance that the MIDlet stores in the `videoControl` field can capture a snapshot once its `Player` instance has begun playback, although it's inadvisable to do so on the main thread, because that can stall either the MIDlet UI or the MMAPI itself. By using the MIDlet instance itself as an explicit lock via the Java `synchronized` keyword and the `Object` method `notify`, the MIDlet can signal from the UI thread to the player thread that you want to capture a frame. In addition, using the `synchronized` keyword ensures that you can't trigger a capture while the application is performing a capture, which is probably an error anyway.

Once the Java VM uses `notify`/`synchronized` to kick the player thread to capture a frame, the player thread performs the capture by invoking the `videoControl`'s `getSnapshot` method, specifying the default encoding. At this point, the player has a `byte[]` of the captured image data; the code just uses the data to set the other image on the `Form` to display the image you captured.

Another thing you might want to do with image data that your application has captured is to share it with someone else. You could do that by changing the `MMSSender` class I show you in Chapter 14 to take an optional `byte[]` of encoded image data, as you see in Listing 16-18, and using the new `MMSSender` method that Listing 16-18 defines.

Listing 16-18. *Revisions to the MMSSender Class to Enable You to Send a Captured Image via MMS*

```
public class MMSSender implements Runnable {
    private byte[] imageBytes;
    public void sendMsg(String r, String id, String m, byte[] bi) {
        if (sending) return;
        receiver = r;
        appId = id;
        msg = m;
        imageBytes = bi;
        encoding = System.getProperty("microedition.encoding");
        Thread t = new Thread(this);
        t.start();
    }

    public void sendMMS() {
        String address = "mms://" + receiver + ":" + appId;
        System.out.println(address);
        MessageConnection c = null;
        try {
            c = (MessageConnection) Connector.open(address);
            MultipartMessage mpm = (MultipartMessage) c.newMessage(
                MessageConnection.MULTIPART_MESSAGE);
            mpm.setSubject("An Image");
            if (image != null) {
                InputStream is = getClass().getResourceAsStream(image);
                imageBytes = new byte[is.available()];
                is.read(imageBytes);
            }
            if (imageBytes != null) {
                mpm.addMessagePart(
                    new MessagePart(
                        imageBytes, 0, imageBytes.length,
                        "image/png", "id1", null, null));
            }
            if (msg != null) {
                byte[] bMsg = msg.getBytes();
                mpm.addMessagePart(
                    new MessagePart(bMsg, 0, bMsg.length,
                        "text/plain", "txt1", null, encoding));
            }
            c.send(mpm);
```

```
        } catch (Exception e) {}
        finally {
            if (c != null) {
                try {
                    c.close();
                } catch (IOException e) {}
            }
        }
    }

…other fields and other methods remain the same from Chapter 14…
}
```

■ **Note** The implementation in Listing 16-18 assumes that the image that the `VideoControl` returns is a PNG image; if you're writing this for a production application, you should extend `MMSSender` to accept the MIME type of the image data you're providing, as well as specify a specific MIME type for the `getSnapshot` method.

Playing SVG Content

Returning to the `MultimediaMIDlet`, the flow for playing SVG content is similar. The `playFromResource` method invokes `initSvgPlayer` and `configViewSvg` and then starts the playback using the resulting `svgAnimator`'s `play` method. Because the SVGAPI's approach to media rendering is a little different than the MMAPI, the flow in `initSvgPlayer` is different, although `configViewSvg` remains an empty method, just as `configViewSound` is. The `initSvgPlayer` method bears a strong resemblance to the pseudocode you first saw in Listing 16-12.

The `initSvgPlayer` method begins by getting an `InputStream` instance to the SVG image in the JAR file, and then creates a new `SVGImage` instance using that data and the default SVGAPI `ExternalResourceHandler`. With the image, the method next creates the `svgAnimator` it will use to play the animation; if the SVG is a static image, that's OK, because it'll just be rendered by the `svgAnimator` instance when `playFromResource` invokes `svgAnimator.play`. Using the `svgAnimator`, the `initSvgPlayer` gets an instance of the `SVGAnimator`'s `Canvas` instance, which it sets aside as the `Displayable` that the MIDlet should show when `playFromResource` sets the next `Displayable`. Finally, this method obtains the default bounds for the `svgCanvas` and uses them to initialize the `svgImage`'s viewport to obtain and set the optimum rendering rectangle for the image on the display.

Wrapping Up

Through optional APIs documented in JSRs 135 and JSR 287, the Java ME platform provides your applications the capability to render rich audio and video in a variety of different formats, including WAV, MPEG, and SVG. Although the MMAPI that JSR 135 defines is fundamentally different than the SVGAPI that JSR 287 defines, together they give you broad latitude in your application's UI design.

The MMAPI uses a paradigm reminiscent of the MVC paradigm many user-interface frameworks, such as Swing, use. Your application obtains or provides a data source to a player that can be controlled by one or more controls that can affect media rendering in some way. The MMAPI provides a `Manager` class, which provides individual `Player` instances given a data source such as an `InputStream` instance or a locator that specifies a data source's location. Locators can point to media on the device, from a sensor on the device, or from off the device on the network; many (but not all) MMAPI implementations support some form of remote media access via HTTP, RTP, or RTSP.

Because the MMAPI provides a Java ME wrapper around dedicated hardware resources, applications that use the MMAPI need to consider carefully when to use those resources. The `Player` interface implements a state machine that helps restrict access to limited resources on the device. A `Player` object can be in one of five states: unrealized, realized, prefetched, started, or closed. The `Player` interface provides methods to transition through these states; typically your application will create a `Player` instance and only invoke its `realize` and `prefetch` methods just before starting playback with `start`.

MMAPI `Player` objects are also factories for `Control` objects; a `Control` may mutate the behavior of a `Player` (such as by adjusting its volume) or may provide additional functionality, such as an interface from which to gain a user-interface component you can use to show video in a media file. Capture from audio and video sensors on a Java device works this way; you specify a locator for the device, and you can use the `VideoControl`'s `getSnapshot` method to obtain an image snapshot, or the `RecordControl` class to record a stream of audio or video data. Not all devices support audio or video capture, however, and the Java ME runtime provides system properties that enumerate precisely what media types and what sensors a specific MMAPI implementation supports. Some devices may support JSR 234, which defines additional `Control` subclasses you can use with the MMAPI to control capture sensors as well as perform additional multimedia operations.

The SVGAPI, on the other hand, supports the SVG Tiny 1.2 standard defined by the W3C. Using SVG, you can define images—static or animated—that appear clear and unpixellated at nearly any rendering size. In fact, you can build whole parts of your application's user interface by specifying events within your SVG document that your Java ME application can receive in response to user operations, such as focus changes. SVG is based on XML; many vector-based drawing programs support this widely adopted standard, making it widely available to mobile content developers. Through its packages and classes, the SVGAPI has a rich set of features, including the ability to access portions of an SVG document through the SVG DOM. In fact, you can even create SVG images on the fly, letting users create new SVG images right on the Java ME device from within your application.

You can use the SVGAPI in one of two ways to render an image: you can use the `SVGImage` class to render static images, or you can use the `SVGAnimator` class to render dynamic images. These work with MIDP `Canvas` subclasses or AWT `Component` subclasses, depending on the Java ME configuration of the target device. It's generally easiest to create an `SVGImage` instance and delegate rendering to an `SVGAnimator`, which has its own rendering thread and state machine to manage animated SVG images as well as any user interaction the SVG image requires. Finally, if you're just knocking out an application using NetBeans and need a simple container to hold an animated SVG (e.g., for a splash screen or interstitial display), NetBeans itself provides a package with some simple components for rendering static or animated SVG images. You can use those components right from the NetBeans GUI builder, making it easy to add SVG-based features to your application.

Regardless of the kind of multimedia your application is going to render, thread management in multimedia applications is crucial. You should never attempt to render multimedia—using either the MMAPI or the SVGAPI—on the main application or MIDlet thread, because the work necessary to perform the rendering may stall the UI thread, making the device feel sluggish or unusable. Both the MMAPI and the SVGAPI provide you with interfaces to schedule operations on the main thread, and you can coordinate activities between the main UI thread and any multimedia threads using simple Java primitives like synchronization via the `synchronized` keyword or a monitor variable.

CHAPTER 17

Finding Your Way

For many developers, the potential of location-based service (LBS) applications is the most exciting aspect of Java ME development today. The potential for new applications that use user position in conjunction with web-available data (for geocoded business locations, locating fellow users, real-time traffic, weather, and other incident information) is simply staggering. Whole new markets are emerging, such as cost-effective, real-time, location-aware navigation; social networking that involves friends' locations; and dispatch and emergency service applications that tap a device's location transparently. JSR 179 defines the Location API, which provides a simple optional API for determining a device's location and sharing location preferences with other applications.

In this chapter, I begin with some brief remarks about LBS for readers who haven't previously encountered the technologies behind it. Next, I introduce the Location API, describe the package and classes that JSR 179 defines for enabling LBS on Java ME devices, and show you how to use the Sun Java Wireless Toolkit to simulate location data when developing and debugging your applications. I close the chapter by showing you an extension to the WeatherWidget application that permits users to specify their current location rather than enter a location when obtaining weather data.

Understanding Location-Based Services

For many years in many countries including the United States, emergency-management officials have been able to determine the location of telephone callers. (In the United States, this is a key part of the 911 system; dialing 911 on any phone connects you to an emergency dispatcher who has information about the location of the phone from which you're calling.) In the mid-1990s, the governments of many countries moved to require carriers to provide the same information for wireless telephone users, giving emergency-response personnel the same capabilities they have when responding to emergency calls from wireline phones. This drove the creation of a number of mobile handset location technologies, including Assisted Global Positioning System (A-GPS) and various cell-tower triangulation systems. Originally intended for use primarily for emergency purposes, network operators were keen to find a means to charge for this service, because providing location-based information on the network incurred high

costs due to the need for hardware changes to their network infrastructure. At the same time, many businesses were quick to recognize the market potential of location-aware wireless terminals for applications as diverse as fleet management and social networking.

The first commercial launch of LBS services was in Japan by KDDI in 2001; early offerings in Japan met with wide acclaim and rapid adoption. Elsewhere, adoption has been slower but is now reaching critical mass as mobile platforms, including Java ME, Qualcomm's Binary Runtime Environment for Wireless (BREW), Apple's iPhone, and Google's Android, provide APIs for LBS applications. In addition, carriers have made the service available for applications on their networks.

Today, several commercially available systems permit wireless devices to determine their location with great precision, provided they have access to one or more Wide Area Network (WAN) services. The most well known is GPS, in which a device triangulates its position using information derived from radio signals received from orbiting satellites. While commercial GPS receivers have been available at low cost for industry and personal purposes for more than a decade, commercial GPS has two major disadvantages that make it ill-suited to be the only means of mobile device positioning. First, GPS receivers require a clear view of the sky in order to be able to receive the radio signals from the satellites that the service uses. Second, GPS accuracy is somewhat variable due to a combination of atmospheric effects and a specific GPS feature named Selective Availability (SA), which permits the US government to compromise the accuracy of civilian GPS in times of war. Fortunately, SA is presently deactivated, and with the widespread civilian adoption of GPS, it is unlikely to be reintroduced, but other errors can lead to an error of several meters. Today, there are ongoing efforts around the world to develop other global navigation assistance systems for political and economic reasons, providing international alternatives to GPS (which works anywhere on the globe).

To address the inherent problems with GPS, several other schemes have been proposed, including A-GPS and various forms of network triangulation. A-GPS works by using additional hints about the device's position gleaned from data about which cell sites it may be using, and it delegates much of the mathematical processing that determining your position requires to network resources. Triangulation techniques rely on determining a device's position based on received signals from local cell towers, and more recently local Wi-Fi base stations as well. To ensure rapid position determination time, as well as high accuracy and availability, today's wireless handsets usually incorporate several schemes for position location.

While the means of determining a device's position may vary under the hood, the result that the positioning system provides is the same: the device's coordinate, which is usually the latitude and longitude computed at a specific time fixed to a *datum*—a mathematical model of the earth's surface. Because the positioning system is approximate—atmospheric effects, radio propagation effects such as multipath, and other factors introduce error into even the best positioning systems—the position that the system provides is a notion of the position's *accuracy*, usually a radius. The coordinates and radius define

a circle in which the device is likely to be. It's important to remember when designing your application that the position data is approximate—even taking into account the position circle. It's possible for a device to fall outside the positioning circle, although in practice this doesn't happen often. This positioning error is often referred to as the *uncertainty* of position, and in some systems, you can trade position uncertainty for other factors, such as the amount of time or financial cost involved in determining the device's position. Depending on device and network capabilities, position information may also be accompanied by the device's elevation, speed, and course (direction of motion), although that isn't always available.

Due to the nature of the positioning systems themselves, obtaining device position is an expensive operation from the perspective of battery power, time, and user cost. Positioning can require scarce device resources as well as network transactions with remote servers, so devices provide position data in near-real time, rather than real time.

Introducing the Location API

The Java ME Location API defines the `javax.microedition.location` package, which contains a collection of classes that permit applications to request and obtain a location result. The Location API was designed to work with both CLDC and CDC devices, although it is only available for CLDC 1.1 and later devices, because the location framework requires floating-point mathematics support in order to provide your application with latitude and longitude information.

The Location API abstracts the device's location subsystem from your application, giving you a way to determine the device's location in a manner that best meets your application's constraints (such as position-acquisition time, uncertainty, and cost to the user). You can request a position just once (often referred to as a *one-shot* positioning request), or you can request that the API deliver notifications of device position while your application is running.

All Location API implementations must provide the latitude and longitude coordinates in a position response, along with the time at which the device determined the position and a measure of accuracy. Depending on the device's hardware, position data may also include the orientation (compass, pitch, and roll) of the device, the speed at which the device is traveling, and more human-palatable data such as a position's street address.

The Location API provides an additional feature that isn't available in most APIs on other platforms (e.g., Qualcomm's BREW), and that's access to a system-wide database of landmarks and the position of those landmarks. This permits the device's native software and all applications on the device to share a single store of common user-specified locations, such as the positions of the user's place of residence and occupation. This enables the user to have a consistent user experience across all LBS applications on the device.

Understanding the Location API

The `javax.microedition.location` package defines nine classes, two interfaces, and two exceptions. Other than the exceptions, which inherit from `Exception`, the class hierarchy is almost completely flat, representing the simplicity of the API and the domain it abstracts. The package provides the following classes:

- `LocationProvider`: Represents a hardware or software module within the device that can determine the device's location. It provides a static factory method that returns specific `LocationProvider` instances configured to respond to specific location requests.
- `Criteria`: Represents specific criteria that apply to a location request, such as the desired accuracy and cost to the user.
- `Location`: Represents a collection of basic information about a location, including a timestamp, coordinates, accuracy, speed, course, and information about the method used to determine the location, as well as an optional textual address.
- `Coordinates`: Represents the latitude, longitude, and altitude of a location, and provides methods to interconvert between floating-point and human-readable textual representations of coordinates.
- `QualifiedCoordinates`: Represents the latitude, longitude, and altitude triple associated with a measurement of the coordinates' accuracy. It is a subclass of `Coordinates`.
- `AddressInfo`: Represents textual information about a location such as the street address. The `AddressInfo` class encapsulates a collection of fields accessed through manifest constants.
- `Orientation`: Represents the physical orientation in space of the terminal, including azimuth (the horizontal pointing direction), pitch (the vertical elevation angle), and roll (the orientation around the device's own longitudinal axis).
- `LandmarkStore`: Represents a store of individual user landmarks. There is always a default landmark store, and implementations may provide additional named landmark stores for application use.
- `Landmark`: Represents a known location with a user-provided name, consisting of the name, a `QualifiedCoordinates` object, and an `AddressInfo` object.

The API defines two interfaces: `LocationListener` and `ProximityListener`. The `LocationListener` gives your application a means to regularly receive position reports from the API implementation through its `locationUpdated` method, and it permits the

API implementation to notify your application if the source of location data has changed (perhaps as a result of transitioning from one network to another). You use the `LocationListener` interface any time you want to receive continuous position notifications, such as an application that provides real-time position reports as the device moves. The `ProximityListener` interface lets your application wait for notification when the terminal has fallen within the proximity radius around a specific coordinate. Using the `ProximityListener` to determine when the device reaches a predetermined location is more efficient than using the `LocationListener`, because the underlying implementation of the positioning hardware may have similar capabilities. It's also easier; you don't need to receive and manage regular position reports just to determine if the device has reached a desired location.

The `LocationException` and `LandmarkException` are exceptions the implementation may throw in response to critical failures of either the location subsystem or the landmark store. For both exceptions, the `String` associated with the exception gives additional information about the details surrounding the failure.

You can do two things with the Location API: determine the device's location, and manage the landmarks the user has created for LBS applications to use.

Using the Location API to Determine Device Location

Using the Location API requires that you perform three steps:

1. Establish the (possibly user-specified) criteria for the location request.
2. Obtain a `LocationProvider` instance that can provide your application with the device's location in a manner that meets your criteria.
3. Determine the position's location.

The pseudocode in Listing 17-1 shows this basic sequence of events.

Listing 17-1. *Determining the Device Location*

```
try {
    Criteria cr = new Criteria();
    cr.setHorizontalAccuracy(10);
    LocationProvider lp = LocationProvider.getInstance(cr);
    Location l = lp.getLocation(60);
    Coordinates c = l.getQualifiedCoordinates();
    if (c != null) {
        // Do something with the location
    }
} catch (LocationException e) {}
  catch (Exception e) {}
```

The Criteria object you create specifies any specific requirements you have regarding the locations your application will request. The Location API implementation uses these to configure and select a specific LocationProvider instance. Table 17-1 shows the specific criteria you can specify within a Criteria object. If the implementation cannot explicitly meet your criteria, it will make its own best-effort selection, provided that the resulting LocationProvider instance meets your specific cost criteria. The Criteria class provides the constants NO_REQUIREMENT, POWER_USAGE_LOW, POWER_USAGE_MEDIUM, and POWER_USAGE_HIGH to describe the case where no criteria apply and to describe specific power-consumption criteria. Listing 17-1 sets a single criterion for the LocationProvider instance: that the location determination fall within ten meters of the probable location of the device.

Table 17-1. *Criteria for Location Provider Configuration*

Criteria	Units	Default Value	Setter	Accessor
Horizontal accuracy	Meters	NO_REQUIREMENT	setHorizontalAccuracy	getHorizontalAccuracy
Vertical accuracy	Meters	NO_REQUIREMENT	setVerticalAccuracy	getVerticalAccuracy
Preferred response time	Milliseconds	NO_REQUIREMENT	setPreferredResponseTime	getPreferredResponseTime
Power consumption	int	NO_REQUIREMENT	setPreferredPowerConsumption	getPreferredPowerConsumption
Cost allowed	boolean	true (permitted to incur cost to the user)	setCostAllowed	isAllowedToCost
Speed and course required	boolean	false	setSpeedAndCourseRequired	isSpeedAndCourseRequired
Altitude required	boolean	false	setAltitudeRequired	getAltitudeRequired
Address required	boolean	false	setAddressInfoRequired	isAddressInfoRequired

The LocationProvider class is a factory of LocationProvider instances, as well as a factory of Location objects and a notifier of location and proximity information. You obtain a specific LocationProvider instance through the class's LocationProvider.getInstance method, passing a Criteria instance that describes the restrictions that should apply to the positioning work the LocationProvider will perform. Once you have

the LocationProvider instance, you can determine either the current position or the last known position (which may be null) by invoking the LocationProvider's getLocation or getLastKnownLocation methods, respectively. Many of the LocationManager methods can throw exceptions—typically LocationExceptions for location-based failures, or SecurityExceptions if the user or platform does not permit the operation.

■ **Tip** Location determination can be a time-consuming process, and the LocationProvider implementation may block the current thread. For this reason, it's best to invoke getLocation in a thread separate from the main thread, to avoid stalling the user interface of your application. I show you how to do this in a real MIDlet later in this chapter, in the "Locating the User" section.

The specific Location instance that the LocationProvider class's getLocation method returns provides the device's location at a specific instance in time that the implementation determined using a specific location method. When you invoke getLocation, you must pass a number indicating the maximum number of seconds your application is willing to wait for the positioning request to complete.

From an instance of Location, you can use the following methods to determine more about the device location:

- isValid: Returns true if the Location is valid with coordinates
- getQualifiedCoordinates: Returns the coordinates of the Location
- getTimestamp: Returns when the location data was collected
- getAddressInfo: Returns the street address (as an AddressInfo object)
- getCourse: Returns the heading in degrees relative to true north
- getSpeed: Returns the device's current ground speed in meters per second at the measurement time
- getLocationMethod: Returns a bit mask describing how the implementation determined the location
- getExtraInfo: Returns any additional information about the request

Table 17-2 describes the bit field values that the getLocationMethod uses to report how the implementation determined the device's location.

Table 17-2. *Location Methods and Location Bit Field Values*

Method	Bit in Bit Mask Asserted	Note
Assisted by other means	`MTA_ASSISTED`	`MTY_NETWORKBASED` and `MTY_TERMINALBASED` indicate assistance source
No assistance used	`MTA_UNASSISTED`	
Angle of arrival	`MTE_ANGLEOFARRIVAL`	Angle of incident radiation on antenna used
Cell ID	`MTE_CELLID`	Cell tower IDs used in determination
Satellite	`MTE_SATELLITE`	Satellite system (e.g., GPS) used
Short range	`MTE_SHORTRANGE`	Bluetooth or other wireless network system used
Time difference	`MTE_TIMEDIFFERENCE`	Time difference between received terrestrial signals used
Time of arrival	`MTE_TIMEOFARRIVAL`	Time of arrival of received terrestrial signals used
Network-based method	`MTY_NETWORKBASED`	Network used in assisting request
Terminal-based method	`MTY_TERMINALBASED`	Position determination used terminal-based method

The Location API provides the `Coordinates` and the `QualifiedCoordinates` classes to represent a specific place in space. A `Coordinates` instance is absolute, specifying latitude, longitude, and elevation, with no error; `QualifiedCoordinates` extends the notion of `Coordinates` by adding information regarding the horizontal and vertical uncertainty in meters. The `Coordinates` class provides methods for obtaining coordinate values (`getAltitude`, `getLatitude`, and `getLongitude`), as well as a generic conversion method (`convert`) for converting between floating-point and string representations of a coordinate. You can also obtain the azimuth and distance between coordinates by invoking one `Coordinates`' `azimuthTo` and `distance` methods, passing each a second coordinate. Because the `LocationProvider` always provides real measurements, it will always return a `QualifiedCoordinate`; you can determine the uncertainty of a specific location measurement using the result's `getHorizontalAccuracy` and `getVerticalAccuracy` methods.

Your application may require more than one location measurement; you can either invoke a `LocationProvider`'s `getLocation` method repeatedly or register an object that implements `LocationListener` to receive location events. Registering a `LocationListener` with a `LocationProvider` via the `setLocationListener` method takes four arguments:

- `listener`: The `LocationListener` implementation that receives the location events
- `interval`: Specifies how often to determine the location and send location events to the listener

- `timeout`: Specifies how late a location may be after the interval elapses
- `maximumAge`: Specifies how old a location result is permitted to be when the event is set (in seconds)

In a similar vein, your application can receive notification when the device is near to a specific location by adding a `ProximityListener` using the `addProximityListener` method. When you do this, you specify the instance that implements the `ProximityListener` interface, the coordinates of the desired location, and a radius in meters around the desired location that defines the region of proximity. Note, however, that while you can have multiple proximity notifications with a single `LocationProvider`, you can only have a single `LocationListener`.

Using the Location API to Manage Landmarks

The Location API includes two classes that let location-aware applications interact with each other through a common store: `Landmark` and `LandmarkStore`. Any LBS-enabled application with security privilege may add to this store, as well as view and modify elements in this store. These two classes enable you to add a landmark in one application—say, a travel guide—and access the same landmark from another application, such as a turn-by-turn navigation application.

As the names suggest, the `LandmarkStore` contains representations of `Landmark` objects. The `LandmarkStore` is persistent, although the actual implementation isn't specified; it may keep its records in an MIDP record store in flash memory or use another storage scheme altogether.

The Location API requires any implementation to have at least one `LandmarkStore`, called the *default* `LandmarkStore`, although the implementation may support additional stores, which you access by name much like an MIDP record store. To access a store, you must first get an instance of `LandmarkStore`, which you do by invoking its static method `getInstance` and passing the name of the store you seek (or `null` for the default store). You can obtain a list of stores by invoking the `LandmarkStore.listLandmarkStores` method, which returns an array of `String`s containing the name of each store on the device. The `LandmarkStore` also provides the static methods `createLandmarkStore` and `deleteLandmarkStore` for creating and deleting a store.

Like the PIM package I showed you in Chapter 7, the Location API supports categories for the `LocationStore`; a category is simply a `String` attached to entries in the store, letting users group related items by that `String` (such as "Work" or "Vacation"). You use the following methods to interact with categories in a store:

- `getCategories`: Enumerates the categories in a store, which returns an `Enumeration`
- `addCategory`: Adds a category
- `deleteCategory`: Deletes a category while retaining the records in the category

Note that you can associate a category with multiple entries in the store, but each entry can be associated with only a single category.

The `LandmarkStore` provides the following methods for adding, deleting, and updating a `Landmark` in the store:

- `addLandmark`: Adds a new landmark to the store
- `deleteLandmark`: Deletes a landmark in the store
- `updateLandmark`: Updates an existing landmark in the store
- `removeLandmarkFromCategory`: Removes a category assigned to a specific `Landmark`

Adding a `Landmark` requires you to pass both the instance of the `Landmark` to add and a category name (or `null` to indicate no category).

You can search the existing `Landmark` instances in a store using one of the three `getLandmarks` methods it defines, all of which return an `Enumeration` of `Landmark`s. Passing no arguments to `getLandmarks` enumerates *all* `Landmark`s in the store. Passing `getLandmarks` a category name (or `null`) and a bounding rectangle of latitudes and longitudes gives you an enumerated list of the `Landmark` instances within that rectangle, while passing just a category (or `null`) and a name returns the `Landmark` instances in the category you specify that possess the name you provide.

A `Landmark` itself is merely a container that holds a user's name, description, address information, and coordinates; the class exposes these through accessors and mutators:

- `getName` *and* `setName`: Gets and sets the `Landmark` name
- `getDescription` *and* `setDescription`: Gets and sets the `Landmark` description
- `getAddressInfo` *and* `setAddressInfo`: Gets and sets the `Landmark` address, which is an instance of `AddressInfo`
- `getQualifiedCoordinates` *and* `setQualifiedCoordinates`: Gets and sets the `Landmark`'s location as an instance of `QualifiedCoordinates`

Understanding the Role That Security Plays in LBS

Many users are understandably *very* sensitive about the interrelationship between location and network services; privacy is often a key concern. Like most optional APIs, the Location API requires privilege on platforms such as MIDP 2.0 that support it. As such, your application may require carrier signing in order to operate on mobile devices once you distribute it. Table 17-3 lists the permission names and protected methods in the Location API. The API provides permissions not just for determining location, but also for accessing the store of landmarks.

Table 17-3. *Location API Permissions*

Permission Name	Methods Protected by This Permission
javax.microedition.location.Location	LocationProvider.getLocation LocationProvider.setLocationListener LocationProvider.getLastKnownLocation
javax.microedition.location.Orientation	Orientation.getOrientation
javax.microedition.location.ProximityListener	LocationProvider.addProximityListener
javax.microedition.location.LandmarkStore.read	LandmarkStore.getInstance LandmarkStore.listLandmarkStores
javax.microedition.location.LandmarkStore.write	LandmarkStore.addLandmark LandmarkStore.deleteLandmark LandmarkStore.removeLandmarkFromCategory LandmarkStore.updateLandmark
javax.microedition.location.LandmarkStore.category	LandmarkStore.addCategory LandmarkStore.deleteCategory
javax.microedition.location.LandmarkStore.management	LandmarkStore.createLandmarkStore LandmarkStore.deleteLandmarkStore

For a user to want to use an LBS application, he or she must trust both the application and the application's vendor. Establishing this trust isn't just about turning on and off various MIDlet permissions like the ones in Table 17-3. When crafting your application, you need to think carefully about why location data is relevant and what risks your users undertake by sharing their location. The value you provide must be greater than the risk—real or perceived—to the application user. In addition, your application should treat location data just as it would any other private data, exposing it only to the degree that is necessary for your application to function. An obvious example would be a social-networking application; a user may well want to share his or her location, but may also want to be able to exercise control over the degree to which it is shared. For example, the user may want to control public display and share in the information only with a circle of specific friends.

Using the Location API

Adding location information to news and weather applications is an obvious use of the Location API, and it's exactly what I do in this section with the WeatherWidget example you've seen throughout the book. Based on the WeatherWidget using the kXML parser from Chapter 13, the WeatherWidget MIDlet shown in Listing 17-2 includes the ability to use your current location as the location for a weather report.

Listing 17-2. *The WeatherWidget Class Using the Location API*

```
package com.apress.rischpater.weatherwidget;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.location.*;

public class WeatherWidget extends MIDlet
    implements CommandListener, Runnable {
    private Form wxForm;
    private Alert locatingAlert;
    private StringItem wxItem;
    private Command exitCommand;
    private Command screenCommand;
    private Command settingCommand;
    private Command okCommand;
    private Command backCommand;
    private Command locateCommand;
    private List locationList;
    private TextBox locationTextBox;
    private Alert cannotAddLocationAlert;
    private WeatherFetcher fetcher;
    private WeatherLocation wxlocation;
    private WeatherLocationStore locationStore;

    private void initialize() {
        locationStore = new WeatherLocationStore();
        String[] locations = locationStore.getLocationStrings();
        try {
            if (locations.length > 0 )
                wxlocation = locationStore.getLocation(locations[0]);
        }
        catch(Exception e){}
        fetcher = new WeatherFetcher(wxlocation, this);
        getDisplay().setCurrent(get_wxForm());
    }

    public void startApp() {
        initialize();
    }
```

```
public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void update() {
    get_wxItem().setText(get_forecast());
    try
    {
        locationStore.updateLocation(wxlocation);
    }
    catch(Exception e){}
}

public void determineLocation() {
    getDisplay().setCurrent(get_locatingAlert());
    Thread t = new Thread(this);
    t.start();
}

public void commandAction(Command command, Displayable displayable) {
    if (displayable == wxForm) {
        if (command == exitCommand) {
            exitMIDlet();
        } else if (command == settingCommand) {
            getDisplay().setCurrent(get_locationList());
        }
    } else if (displayable == locationList) {
        if (command == locateCommand) {
            determineLocation();
        } else if (command == screenCommand) {
            getDisplay().setCurrent(get_locationTextBox());
    } else if (command == List.SELECT_COMMAND) {
            int index = get_locationList().getSelectedIndex();
            set_location(get_locationList().getString(index));
            fetcher.cancel();
            fetcher = new WeatherFetcher(wxlocation, this);
            getDisplay().setCurrent(get_wxForm());
    } else if (command == backCommand) {
            getDisplay().setCurrent(get_wxForm());
        }
```

```
        } else if (displayable == locationTextBox) {
            if (command == locateCommand) {
                determineLocation();
            } else if (command == backCommand) {
                getDisplay().setCurrent(get_locationList());
            } else if (command == okCommand) {
                add_location(locationTextBox.getString());
                getDisplay().setCurrent(get_locationList());
            }
        } else if (displayable == cannotAddLocationAlert) {
            if (command == backCommand) {
                getDisplay().setCurrent(get_locationList());
            }
        }
    }

    public String get_forecast() {
        if (wxlocation == null) {
            return "unknown forecast";
        } else {
            return wxlocation.getForecast();
        }
    }


    public String get_location() {
        if (wxlocation == null) {
            return "unknown";
        } else {
            return wxlocation.getLocation();
        }
    }

    public void set_location( String l ) {
        try {
            wxlocation = locationStore.getLocation(l);
        }
        catch(Exception e) {}
        get_locationTextBox().setString(l);
        get_wxForm().setTitle(l);
    }
```

```
public void add_location( String l ) {
    wxlocation = new WeatherLocation(l);
    try {
        locationStore.addLocation( wxlocation );
    } catch (Exception e) {
        getDisplay().setCurrent(get_cannotAddLocationAlert());
    }
    locationList = null;
}

public Display getDisplay() {
    return Display.getDisplay(this);
}

public void exitMIDlet() {
    getDisplay().setCurrent(null);
    destroyApp(true);
    notifyDestroyed();
}


public StringItem get_wxItem() {
    if (wxItem == null) {
        wxItem = new StringItem("Forecast", get_forecast());
    }
    return wxItem;
}

public Form get_wxForm() {
    if (wxForm == null) {
        wxForm = new Form(get_location(), new Item[] {
            get_wxItem()
        });
        wxForm.addCommand(get_exitCommand());
        wxForm.addCommand(get_settingCommand());
        wxForm.setCommandListener(this);
    }
    return wxForm;
}
```

```
    public Alert get_locatingAlert() {
        if (locatingAlert == null) {
            locatingAlert = new Alert("Locating", "Finding your location",
                    null, null);
            locatingAlert.setTimeout(Alert.FOREVER);
        }
        return locatingAlert;
    }

    public TextBox get_locationTextBox() {
        if (locationTextBox == null) {
            locationTextBox = new TextBox("Add Location", "", 80, 0);
            locationTextBox.addCommand(get_locateCommand());
            locationTextBox.addCommand(get_backCommand());
            locationTextBox.addCommand(get_okCommand());
            locationTextBox.setCommandListener(this);
        }
        return locationTextBox;
    }

    public List get_locationList() {
        if (locationList == null) {
            String[] locations;
            locations = locationStore.getLocationStrings();

            locationList = new List("Where", List.IMPLICIT, locations, null);
            locationList.addCommand(get_screenCommand());
            locationList.addCommand(get_backCommand());
            locationList.addCommand(get_locateCommand());
            locationList.setCommandListener(this);
        }
        return locationList;
    }

    public Alert get_cannotAddLocationAlert() {
        if (cannotAddLocationAlert == null)
        {
           cannotAddLocationAlert = new Alert("Cannot Add Location");
           cannotAddLocationAlert.setString("An error occurred adding➥
the location you entered. It has not been added.");
           cannotAddLocationAlert.addCommand(get_backCommand());
        }
        return cannotAddLocationAlert;
    }
```

```
public Command get_settingCommand() {
    if (settingCommand == null) {
        settingCommand = new Command("Settings", Command.OK, 1);
    }
    return settingCommand;
}

public Command get_okCommand() {
    if (okCommand == null) {
        okCommand = new Command("OK", Command.OK, 1);
    }
    return okCommand;
}

public Command get_locateCommand() {
    if (locateCommand == null) {
        locateCommand = new Command("Use this location", Command.ITEM, 1);
    }
    return locateCommand;
}


public Command get_exitCommand() {
    if (exitCommand == null) {
        exitCommand = new Command("Exit", Command.EXIT, 1);
    }
    return exitCommand;
}

public Command get_screenCommand() {
    if (screenCommand == null) {
        screenCommand = new Command("Add Location", Command.SCREEN, 1);
    }
    return screenCommand;
}


public Command get_backCommand() {
    if (backCommand == null) {
        backCommand = new Command("Back", Command.BACK, 1);
    }
    return backCommand;
}
```

```
    private class LocationUpdater implements Runnable {
        String location;
        public LocationUpdater(String l) {
            location = l;
        }
        public void run() {
            add_location(location);
            set_location(location);
            getDisplay().setCurrent(get_locationTextBox());
        }
    }

    public void run() {
        Criteria criteria = new Criteria();
        LocationProvider lp;
        StringBuffer cl = new StringBuffer();
        String location;

        criteria.setCostAllowed(true);
        criteria.setAddressInfoRequired(true);
        criteria.setHorizontalAccuracy(100);
        criteria.setVerticalAccuracy(100);

        try {
            lp = LocationProvider.getInstance(criteria);
            Location l = lp.getLocation(60);
            if ( l != null && l.isValid() ) {
                AddressInfo ai = l.getAddressInfo();
                QualifiedCoordinates c = l.getQualifiedCoordinates();
                if ( ai != null ) {
                    cl.append(ai.getField(AddressInfo.CITY));
                    cl.append(", ");
                    cl.append(ai.getField(AddressInfo.STATE));
                    cl.append(", ");
                    cl.append(ai.getField(AddressInfo.COUNTRY));
                } else {
                    int r = QualifiedCoordinates.DD_MM;
                    String s;
                    cl.append("GPS Coordinates, ");
                    s = QualifiedCoordinates.convert(c.getLatitude(), r);
                    cl.append(s);
                    cl.append(", ");
```

```
                    s = QualifiedCoordinates.convert(c.getLongitude(), r);
                    cl.append(s);
                }
            }
        } catch(Exception e) {};
        if ( cl.length() > 0 ) {
            location = cl.toString();
        } else {
            location = "Could not determine location.";
        }

        getDisplay().callSerially(new LocationUpdater(location));
    }
}
```

The core UI implementation remains the same, but I made a few changes for clarity. The most obvious change is that I renamed the `Location`, `LocationStore`, and `LocationParser` classes to `WeatherLocation`, `WeatherLocationStore`, and `WeatherLocationParser`, respectively. This isn't strictly necessary, as I could resolve the naming collision between my use of `Location` as a class name throughout the book and the Location API's `Location` class using package qualifiers in the `WeatherWidget` class and elsewhere, but doing so may be confusing.

To support the location features, I added the `locatingAlert` to provide status while the MIDlet uses the Location API. I also added the `Command` instance `locateCommand` to the `locationList` and `locationTextBox` screens. Using the same scheme for organizing UI code that NetBeans imposes, I added lazy constructor-fetcher methods for these items. I could have added them directly using the NetBeans GUI builder, but small changes like this frequently go faster if you edit the code by hand.

■ **Caution** Choosing whether to rely on the NetBeans GUI builder or hand-build GUIs (in whole or in part) is a difficult decision. I find that using NetBeans to lay out my GUI gives me a way to rapidly prototype an application's flow, although it's sometimes faster to make small tweaks by hand. For this book, I chose to use the GUI builder for initial examples and then hand-edit those examples to minimize the amount of additional code that gets added to a project (either by me or NetBeans itself). Doing this can be dangerous, because the NetBeans code generator doesn't like you changing its autogenerated code. In practice, it's best to either rely on a code-generation tool or build code by hand, and not mix the two approaches.

To show you the location that the MIDlet determines with the Location API, I tweaked the implementation of the `set_location` method slightly. In addition to updating the MIDlet's `wxlocation` field, the `set_location` method also sets the `locationTextBox`'s contents to the location the MIDlet has determined with the API.

The `WeatherWidget` uses a separate thread to obtain your position, although the code for that thread is within the `WeatherWidget` class itself rather than a separate class. The `determineLocation` method creates and starts this thread in response to you choosing "Use this location" from the list of locations or the location-entry screen. As this thread finishes execution, it schedules a `Runnable` task on the main thread to add the new location to the list of locations, and it sets the MIDlet to show the `locationTextBox` so you can see the results of the location request.

Locating the User

The MIDlet uses a one-shot positioning request to determine your location; the location thread does this work when you invoke `determineLocation` by spawning a new thread that executes the `run` method in Listing 17-2. This code is an extension of the pseudocode you saw in Listing 17-1; it has only a few important differences.

The application sets specific criteria for the location request, indicating that it's OK for the location request to incur cost, that an `AddressInfo` should accompany the request, and that it's OK to set a relatively high margin of error (100 meters). The `run` method validates the resulting `Location` instance, using the resulting `AddressInfo` structure if it's available to provide the `WeatherLocation` structure with a real city, state, and country, just as if you'd entered this information directly. In most cases, however—and certainly in the Sun Java Wireless Toolkit—all that you receive is a `QualifiedCoordinates` instance, and the code just uses that instead, building up a `String` that consists of the tag `GPS Coordinates` followed by the latitude and longitude. This is admittedly a hack, but a reasonable one: it uses the existing implementation of the other classes in the MIDlet and permits me to focus on showing you the Location API, rather than how to refactor the other classes to work with either coordinate data or its human-usable counterpart.

Once `run` builds up the `String` containing the location data, it uses a separate `Runnable` to update the UI on the main thread. This isn't strictly necessary, because the `locatingAlert` obstructs the `locationTextBox`'s input anyway, so there's no possibility for conflict between the two threads. However, it demonstrates to you how to schedule an action on the main UI thread. The private inner class `LocationUpdater` actually performs the work necessary to update the UI, which could have been an anonymous class, since it only occurs here. Its constructor takes the new location, while its `run` method adds the new location to the location list, sets the current location to the new location, and changes the display from the `locationAlert` to the `locationTextBox`, which shows the determined location.

Simulating Location API Data in the Sun Java Wireless Toolkit

The Sun Java Wireless Toolkit supports the Location API in simulation; you can specify either a fixed location or a script of locations that the emulator will use over its lifetime. Figure 17-1 shows the configuration screen (select MIDlet ➤ External Events from the emulator's main menu, and choose the Location tab).

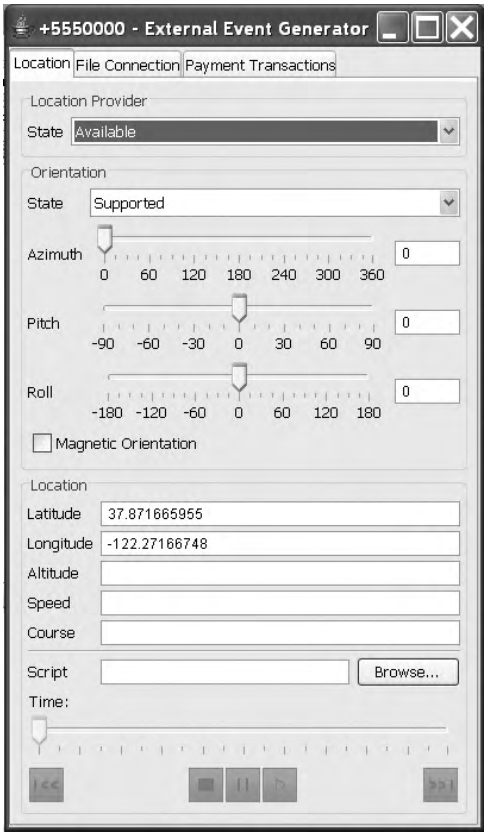


Figure 17-1. *Simulating the device location*

As you can see from the figure, you can choose to simulate not just the device position, but also its orientation, altitude, speed, and course. You can also simulate location failures by changing the Location Provider combo box, letting you test your application in a predictable and repeatable way.

While specifying your position manually is a good way to test simple applications that use one-shot positioning, you'll want to vary the simulated position over time when testing more complex applications (especially those that use multiple location requests or proximity detection). You can do this using a simple script in XML that includes the following two XML tags:

- `waypoints`: This top-level tag contains a list of individual `waypoint` items, each a separate position.

- `waypoint`: This tag contains a specific position and timestamp using the time (in milliseconds), latitude, longitude, and altitude (in meters) attributes.

Listing 17-3 shows a simple example.

Listing 17-3. *A Simple Script for Simulating Motion in the Sun Java Wireless Toolkit*

```
<?xml version="1.0"?>
<waypoints>
  <waypoint time="0"
            latitude="37.87" longitude="-122.22" altitude="10" />
  <waypoint time="50000"
            latitude="38.0" longitude="-122.30" altitude="12" />
</waypoints>
```

By clicking the Browse… button and selecting a script with `waypoint` elements, you can then replay the position data within the Sun Java Wireless Toolkit using the playback controls at the bottom of the Location pane in the Location tab.

■ **Tip** Need a lot of test data? GPS receivers with USB or serial ports are cheap, and nearly all of them output data in a standard format dictated by the National Marine Electronics Association (NMEA). Using one, you can hike, drive, or bicycle around town and capture position information on a laptop computer, then convert the data to a list of `waypoint` entries to use in testing your application. NMEA uses a comma-delimited, plaintext format; converting the file to use with the Sun Java Wireless Toolkit can be as simple as writing a quick Emacs macro or a little Java command-line tool.

Wrapping Up

The optional Location API provides a general wrapper around the wide number of positioning technologies that devices use today. Contained within the `javax.microedition.location` package is a `LocationProvider` class that permits you to obtain the device's location either once or as a sequence in time; you can also use it to obtain an event when the device approaches some location.

The `LocationProvider` class is a factory for both `LocationProvider` and `Location` instances, as well as a notifier of `locationUpdated`, `proximityEvent`, `providerStateChanged`, and `monitoringStateChanged` events. To use the interface, you obtain an instance of `LocationProvider` through its `getInstance` static method, passing a `Criteria` object that describes your requirements for the location information you seek. You can then register `LocationListener` or `ProximityListener` implementations with the `LocationProvider` instance to receive continuous location information or proximity events, or simply determine the location of the device as needed by invoking its `getLocation` method. All of this should be done on a separate thread, of course, to avoid stalling the UI thread.

The API represents location data as instances of the `Location` class, which bear fields representing the coordinates, street address, timestamp, course, speed, and means of location for the instance. To standardize access to this data, the API also includes classes such as `AddressInfo` and `QualifiedCoordinates`, which represent a street address and location with an associated error circle, respectively.

The API also permits different applications (potentially including the embedded applications on the device) to share location data with each other through one or more landmark stores. You access the landmark stores through the `LandmarkStore` class, which provides static methods for creating and removing stores from the persistent file system as well as accessing a specific store. Once you create an instance of `LandmarkStore`, you can use it to enumerate the contents of a store, obtaining individual `Landmark` objects that represent a specific `Location` instance and user-supplied data such as the name, description, and category for the instance.

CHAPTER 18

Seeking a Common Platform

The balkanization of the Java ME platform is undoubtedly its weakest point. As you've seen in the chapters throughout this part, a number of optional APIs enable you to create compelling applications for the platform. Unfortunately, it's often difficult to determine which devices support which of these APIs, and if you use more than a few optional APIs in your application, you'll need to manage the combinatorial explosion that results from the number of different optional APIs that may or may not be on target devices in the market. This fragmentation is an undesirable side effect of the JCP; as vendors attempt to extend Java ME to support their device's features, they propose new extensions to the Java ME platform. The JCP works reasonably well in ensuring that when different vendors differentiate products, the new features are available via a common API. However, it suffers from the presence of too many optional APIs from which to choose. Many organizations active in the JCP recognize this problem and have proposed several JSRs to address this problem.

In this chapter, I discuss the Java ME fragmentation problem in more detail, helping you understand both why it occurs and what you can do to plan around it in your development efforts. I talk more about the actual contents of the typical JSR, because for you to become truly fluent as a Java ME developer, you need to be able to understand what a JSR does and does not provide. I then examine in detail the three key JSRs that address Java ME fragmentation today: JSR 185, which defines the Java Technology for the Wireless Industry (JTWI); JSR 248, which defines the Mobile Service Architecture (MSA) 1.0; and JSR 249, which defines MSA2.

Understanding the Role JSRs Play in Fragmentation

As you saw in Chapter 1, Java ME's roots as a platform for small computing devices drove its creators to take a minimalist, lowest-common-denominator approach in which the basic platform would contain little functionality. By adding new modules to the platform,

Sun and other vendors could provide additional features in a predictable and consistent manner. The introduction of configurations, profiles, and packages lets Sun provide versions of Java ME that meet the capabilities of specific classes of devices, and it lets developers target those devices.

Contributing to Fragmentation and Unification

Sun licenses Java ME to an open market of platform and device vendors, many of whom want to offer additional software or hardware features to differentiate their products. The JCP addresses this by letting competing vendors work together to describe extensions to Java ME (and Java as a whole). For example, prior to the FCOP, a number of handsets on a few US carriers had proprietary, file system–access interfaces based on Java SE. While nothing was architecturally wrong with these interfaces, the very fact that one carrier supported handsets from multiple vendors with different interfaces to access files on the file system was a big problem: if you wanted to achieve a wide market share on that carrier with an application that required file-system access, you needed to write and distribute multiple versions of your application. The FCOP addressed this by providing a single API that all platform vendors offer. Now you only need to write your code using the API that FCOP provides in order to run on any device that provides file-system access.

This solution works well for single API sets, such as the FCOP or the MMAPI, but falls short when addressing the fact that different device vendors offer differing sets of optional APIs. If you're writing a single application that requires only one or two optional APIs, this may not be a large issue for you, but increasingly complex mobile applications require the support of more than one or two optional APIs. These applications are often at risk of not being able to gain the market share that can support their development, simply because there's little clarity regarding which optional APIs platform vendors will *actually* support out of the plethora of those that exist.

An obvious solution to the problem is to apply the configuration/platform/package model that Sun uses to describe Java ME versions to the optional packages, defining new configurations and platforms. In essence, this is exactly what JCP members have done through other JSRs. Some JSRs—such as the ones I'm discussing in this chapter, which are JSR 185, JSR 248, and JSR 249—don't describe optional APIs; instead, they describe collections of optional APIs and provide additional clarification as to the acceptable behavior of specific optional APIs where necessary. In essence, these JSRs attempt to unify the platform through the prescription that a set of optional APIs be required in order for a particular name or label to apply. For example, as you'll see in the next section, "Understanding the JTWI," a JTWI-compliant device described by JSR 185 must include CLDC 1.0, MIDP 2.1, and WMA 1.1; it may also include MMAPI 1.1. Device vendors can—and should—choose to meet the JTWI standard in order to make their products palatable to the market, and carriers can require that devices meet the JTWI standard prior to permitting them on their network.

Admittedly, this solution suffers from the same problem that it's trying to cure: there's no set individual or organization that can actually dictate to vendors exactly which JSRs they must support. Consequently, these attempts at unification are themselves not perfect; they rely on market forces for their success. However, the expert groups behind these JSRs often include key manufacturers and operators—companies including Nokia, Vodafone, Motorola, Orange, and Sun—lending credibility and momentum to these efforts to stem platform fragmentation.

Reading a JSR

Since you've made it this far in the book, I'll let you in on a little secret: a lot of what I've told you throughout this book is available in the JSRs that document the APIs you've read about.[1] In fact, Java ME changes so quickly that in practice, it's better for you to be familiar with the JSRs that define the platform than for you to have simply read a book or two on Java ME. (Of course, there's no substitute for actual experience writing code for the platform, too!) That way, as vendors introduce new APIs for emerging technologies, you'll be prepared to learn about the APIs and how to use them in your application.

Broadly speaking, most JSRs have the following contents (often with these titles):

- *Introduction*: States the purpose for the JSR. This introduction usually tells you the business case for the JSR; in the case of an optional API, it will tell you why the expert group that drafted the JSR felt the Java developer community needed the optional API.
- *Contributors*: Lists the companies (and often people) behind the creation of the JSR.
- *Glossary*: Defines domain-specific terms that the JSR uses, and often recaps the definitions of *must*, *must not*, *should*, *should not*, and *may*, as described in Request for Comments (RFC) 2119, "Key Words for Use in RFCs to Indicate Requirement Levels."
- *Packages*: Describes the packages the JSR introduces to the Java environment.
- *APIs*: Describes the classes and interfaces in the packages the JSR introduces.
- *Samples*: Shows how to use the APIs the JSR introduces. If you're lucky, the JSR will contain samples.

1. That said, I've tried to give you practical experience in applying the Java ME APIs, as well as make the material a little less dry than Atacama Desert.

Many JSRs actually contain the full package documentation in HTML from Javadoc; some JSRs, such as JSR 135, provide *all* of their information in that form. Consequently, to use the information in a JSR successfully, you need both some domain experience about the problem the JSR is trying to solve, as well as a clear understanding of the packages the JSR defines and the classes those packages contain.

JSRs themselves have a clear life cycle, which includes the following phases:

1. *Draft*: JSRs frequently undergo several early drafts, many of which are reviewed privately by the expert group that puts forth a JSR. During this phase, you may not even know that the JSR exists; if you do, you must recognize that its contents will almost certainly change at some point in the future.

2. *Public review*: During this time, anyone may see or comment on a JSR by addressing comments to the expert group responsible for the JSR. This phase ends during an approval ballot conducted among members of the JCP; if a document does not pass the approval ballot, the expert group may submit a revised version that addresses the community's concerns. Once a JSR has passed the public review stage, it's likely to undergo little change in the future, except for maintenance.

3. *Maintenance*: Once a JSR is accepted by the community, the expert group providing the JSR may make minor changes to it to address items that need correction.

4. *Dormancy*: A JSR that no longer has an expert group behind it is said to be dormant. It's likely that dormant JSRs describe APIs that do not have widespread support among the community, or APIs that have been clearly superseded by newer initiatives.

Understanding where a JSR is in its life cycle can help you determine whether or not you can depend on the APIs it provides in your product planning. For example, basing a product on an API described in a JSR in the early draft phase may be dangerous. Before basing a product on a dormant JSR, it's a good idea to understand why that JSR is dormant.

Over time, I've come up with an approach that enables me to learn and work with the information in a JSR fairly quickly. The trick is to not only look at the APIs, but also to try to gain some domain experience about the problem the API designers are trying to solve. For example, when looking at the SATSA-CRYPTO API in JSR 177, I spent some time looking at the various technologies in the smart-card industry, even though the project I was working on wasn't likely to use a specific vendor. I also spent some time looking at other APIs that other communities have developed that meet the same objective; seeing how different people approach the same problem of abstracting access to a resource can shed light on the decisions made for the implementation that the JSR actually documents. Always keep in mind that at some level, the APIs a JSR defines are really just an abstraction of some physical system; it's easy to get so wrapped up in the class and object model that you forget the real-world underpinnings.

Another thing that can help you understand a JSR is to draw class diagrams for the interfaces and classes the JSR's packages provide—even if the JSR authors helpfully provide the diagrams anyway. If you can read through the description of a set of classes and methods and produce a diagram of the relationships between those classes and methods, you're well on the way to actually understanding how they work together.

A word of warning is in order, however. JSRs are in English, a natural language fraught with nuance and uncertainty. Like any software system specification, it's difficult to eliminate ambiguity when working with natural languages; worse, JSRs are often meant to be descriptive of a number of actual implementations, and often aren't prescriptive enough to cover all the possible ambiguities. JSR 135, which describes the MMAPI, is a painful illustration; you need only consider the number of caveats I offered in Chapter 16's "Capturing Media" section to realize that the JSR isn't the whole story. Some JSRs actually try to address the shortcomings of other JSRs; the two JSRs describing the MSA provide an example of JSRs attempting to mend flaws in previous JSRs.

Sun (and occasionally other companies) provide reference implementations (RIs) to vendors for key packages, including the MMAPI. The RI is often the implementation of the package in one or more JSRs that finds its way into the Sun Java Wireless Toolkit and corresponding NetBeans Mobility Pack; Sun may also make the RI available to vendors as a starting point on which to base their implementation. Additionally, Sun also makes Technology Compatibility Kits (TCKs) available for many of these key optional APIs; vendors can use the TCK's tests as a way to determine the degree to which *their* implementation matches the RI. Of course, all of this assumes that the RI is a perfect implementation of a perfect JSR that has no ambiguity (in either the eyes of the RI implementors *or* you!)—a rather unlikely confluence of events in today's software world.

The solution, of course, as I've urged you throughout this book, is to implement and test your assumptions about any API you use. Although it's extremely rare for an API or a tool to be broken or nonexistent, it's entirely possible that some element of the documentation around an API will not be quite clear (either to you or to someone who based his or her implementation on the documentation). If you're learning a new API, building a prototype of an application that uses key parts of that API is a good way to both cement your understanding of the API and determine if it's as solid as it ought to be. Furthermore, you can use that same prototype on different device targets to explore any nuances of the device implementation of the API. As in most things, there's really no substitute for experience.

Dealing with Fragmentation on Your Own

You may find yourself in the uncomfortable position of having to develop an application using APIs that have no JSR; worse, the APIs may be different on different platforms. This can happen, especially when working closely with device manufacturers or when relying on cutting-edge features that the community just hasn't reached unity on. In that case,

one thing you can do is provide an isolation layer that abstracts the nonstandard API, and implement to that isolation layer. This lets you swap in and out implementations of the isolation layer that conform to specific API implementations, so that as you port your code from one nonconforming device to another, you minimize the changes in your application to the nonstandard API.

If the interface consists solely of logic, you can also make a reference implementation yourself; for example, consider how you could use the kXML parser on devices that do not support XML parsing through JSR 172. Of course, you may be able to do this to make up for deficiencies in devices that don't provide implementation for a specific API, too. This approach can also help you when working with the Sun Java Wireless Toolkit, because you may not be able to run your application without the optional API you're missing in emulation, making it more difficult to develop and debug your application.

You should also consider becoming part of the solution by joining the JCP and participating in the standards process yourself.

Understanding the JTWI

The JTWI, defined by JSR 185, was one of the first attempts to unify the optional APIs available for Java ME devices—specifically, mobile phones. Finalized by 2003, it actually requires very little above and beyond the CLDC/MIDP stack that most Java-enabled mobile phones were including at the time. It was a crucial milestone, however, in that it set out some clear goals for Java ME implementations of mobile phones.

On the hardware front, the JTWI makes some strong recommendations, the following being chief among them:

- *Color screen*: The device must have a minimum screen size of 125×125 pixels, with a pixel aspect ratio of 1:1. The screen must be color, with a color depth of at least 12 bits.

- *Heap*: The device must have at least 256KB of heap available to the Java VM.

Thus, while it's *possible* you'll encounter a JTWI device with a smaller screen or heap, it's extremely unlikely.

JTWI devices must meet some rigid specifications, including the following:

- *JAR file size*: The AMS must be able to support JAR files of at least 64KB.

- *JAD file size*: The AMS must be able to support JAD files of at least 5KB.

- *Record store:* The record store implementation must be able to store at least 30KB.

- *Threads*: The Java VM must permit a MIDlet suite to create a minimum of 10 simultaneously running threads.

- *JPEG support*: The `Image` class must be able to support JPEG images as well as PNG images required by MIDP 2.0.

- *GSM SMS*: The WMA implementation must support GSM SMS, including the ability to deliver SMS messages to MIDlets via the push registry.

- *HTTP 1.1*: Each device must provide HTTP 1.1 for all supported media types, as required by MIDP 2.0.

In addition, the JTWI requires CLDC 1.0 or 1.1, MIDP 2.0, and WMA 1.1. Vendors may include MMAPI 1.1 as well.

Examining the JTWI Required Elements

To reach the lowest tier of devices (and thereby encompass a large number of target devices), the JTWI requires relatively few packages. However, to ensure compatibility between those devices, the JTWI requires the following elements:

- *CLDC*: The JTWI requires CLDC 1.0, as defined in JSR 30. Vendors may substitute CLDC 1.1, as defined in JSR 139, because CLDC 1.1 is a strict superset of CLDC 1.0. The CLDC—either 1.0 or 1.1— provides a robust core foundation upon which all applications depend (see Chapter 2).

- *MIDP*: The JTWI requires MIDP 2.0, as defined in JSR 30. This provides the UI, persistent store, and communications layer that many applications require (see Chapters 2, 3, 4, and 5).

- *WMA*: The JTWI requires WMA 1.1, as defined in JSR 120—including SMS—to enable applications to take advantage of the wireless network on which JTWI-compliant devices operate (see Chapter 14).

The JTWI provides some clarification for each of these JSRs, as I note in the previous section. The curious (or excessively pedantic) should refer to JSR 185 for details.

Examining the JTWI Optional Elements

At the time JSR 185 was written, multimedia interfaces were just becoming available on some mid- and high-tier mobile handsets. To encourage adoption of the MMAPI (which I described in Chapter 16) where it can be made available, the JTWI requires that if a device supports multimedia access through Java ME, it be made available through MMAPI 1.1. If an MMAPI implementation exists on a JTWI device, the device must support the following features:

- *HTTP:* The device must support HTTP 1.1 download (*not streaming*) of all supported media formats. This is because the MMAPI itself does not support any mandatory protocols, nor does it specify which media types require a specific protocol.

- *MIDI*: The device must support MIDI file playback. This is because many games are enhanced by suitable audio; MIDI provides a compact representation of that audio. In addition, the device must also support the `VolumeControl` for MIDI playback.

- *Tone sequences*: The device must support tone sequences as an additional route to provide audio for games and other applications.

- *Snapshot format*: If the MMAPI implementation supports the video feature set and video image capture, the snapshot format must include JPEG encoding as an encoding option.

Frustratingly, there's a lot that the JTWI *doesn't* say about the MMAPI, such as whether it should support streaming and digital audio formats such as WAV or MP3. By setting the bar as low as it does, though, it includes many commercially available devices, and it has helped the Java ME developer community standardize many aspects of multimedia development for wireless games.

Understanding the MSA

The JTWI is an excellent first step in unifying Java ME APIs for the mobile-phone market. However, it is seriously dated; the fast pace of hardware and software development for mobile devices guaranteed that by the time the JTWI would be widely adopted and referenced, it would not address further fragmentation from additional features such as the FCOP, additional MMAPI and SVG support, and other features such as Bluetooth and the Mobile 3D Graphics API, neither of which I have even discussed at any length this book. Consequently, the Java ME community has produced JSRs 248 and 249 to define the MSA—a platform architecture more broad than the one that the JTWI defines, yet still targeted at high-volume devices. Moreover, while the JTWI largely captured the state of the Java ME device market at the time of its acceptance, the MSA aims to describe not only features that are already available on some devices, but also those that experts expect will be prevalent in the foreseeable future.

There are already two versions of the MSA: MSA 1.0, which JSR 248 defines; and MSA2, which JSR 249 defines. MSA 1.0 is fully accepted by the JCP; the maintenance release of JSR 248 was released as I wrote this chapter. By comparison, JSR 249 is a wee little thing; while its predecessor JSR 248 was being approved, JSR 249 was in private draft

state and not even visible to the entire community. However, it's worth discussing both, because it's not clear which the market will eventually adopt, and both provide an excellent framework for discussing the immediate future of the Java ME platform.

Like the JTWI before it, the MSA defines a collection of APIs that conformant devices must support, and it provides specific clarifications regarding ambiguities surrounding many of those APIs. Together, MSA 1.0 and MSA2 define five different permutations of optional Java ME APIs for mass-market phones using technologies and APIs readily available either now or in the near future.

Understanding MSA 1.0

The efforts to craft MSA 1.0 began some two years after the completion of the JTWI, and you can see this clearly when you look at the number and types of optional interfaces it requires for conformance. The JSR defining MSA 1.0 specifies mandatory components and requirements that must be fulfilled, as well as *conditionally mandatory* requirements that must be fulfilled if specific conditions set out in the JSR are met.

The JSR contains five key sections that describe the mandatory component JSRs, additional clarifications about component JSRs, additional platform requirements above and beyond the JSRs, various recommendations for developers, and a roadmap detailing a tentative future for Java ME and the MSA. The expert group behind the specification consists of major players in the Java ME market, including hardware manufacturers such as Motorola, Nokia, Research In Motion, Samsung, Siemens, and Sony, as well as network operators including Cingular Wireless (now part of AT&T Mobility), NTT docomo, Orange, Sprint, T-Mobile, and Vodafone, as well as of course Sun. The specification defines two platforms: *MSA* and *MSA Subset*. (To avoid confusion with MSA2 that JSR 249 defines, I'll refer to these as MSA 1.0 and MSA Subset 1.0.) As the name suggests, MSA 1.0 is a full software stack, while MSA Subset 1.0 eliminates several optional software packages for lower-cost, less-powerful devices. In order to meet the requirements of either of these platforms, a compliant implementation must implement or comply with

- *JSRs*: It must implement every JSR indicated as a required component for the platform as specified by MSA 1.0 or MSA Subset 1.0
- *Additional clarifications for JSRs*: It must comply with any additional MSA 1.0 or MSA Subset 1.0 requirements as detailed in the JSR's relevant "Additional Clarifications" section
- *Additional requirements:* It must comply with all of the additional MSA requirements set out in JSR 248

Some requirements are common to both MSA 1.0 and MSA Subset 1.0, including requirements in the following categories:

- *JTWI*: An MSA-compliant implementation must meet the requirements of the JTWI.
- *Screen*: An MSA-compliant implementation should have a screen with at least 128×128 pixels with 16 bits of color supporting 65,536 colors. (It's possible you'll encounter one that doesn't, but this is unlikely.)
- *Heap*: The minimum permitted Java heap available to a MIDlet should be at least 1024KB; the MSA recommends that the heap should be at least 2048KB.
- *JAD and JAR file restrictions*: The minimum supported MIDlet JAR size is 300KB. The minimum supported MIDlet suite JAD file is 10KB. The platform must support JAD manifests with up to 512 attributes.
- *Record store*: The minimum supported record store size per MIDlet suite is 64KB, and each MIDlet suite may use a minimum of 10 separate record stores.
- *Security*: The MSA defines a clear correspondence between optional package APIs requiring permission and the name of the relevant permission.
- *System properties*: The MSA defines and provides a common reference for the system properties that you use to determine the availability and version of specific APIs (e.g., the FCOP).
- *Network protocols*: HTTP, HTTP over Secure Sockets Layer (SSL) 3.0, TCP, SMS, and MMS are mandatory protocols that MSA implementations must provide. In addition, MSA 1.0 (not MSA Subset 1.0) must provide support for the Session Initiation Protocol (SIP). Other protocols, such as HTTP over TLS, TCP server support, TLS 1.0, and UDP, are optional. Some protocols, including several profiles of Bluetooth and Infrared Data Association (IrDA), are conditionally mandatory.
- *Content formats*: MSA requires support for a number of common content formats, including PNG, JPEG, the Mobile 3D Graphics API, SVG Tiny 1.1, WAV in 8kHz pulse-code modulation (PCM) format, Adaptive Multi-Rate Narrow Band (AMR-NB) audio compression, MIDI, and Scalable Polyphony MIDI (SP-MIDI) audio.

Examining MSA Subset 1.0

To be considered MSA Subset 1.0–compliant, a product must provide the following packages:

- *CLDC 1.1 and MIDP 2.1*: Provide the foundation platform as specified in JSR 139 and JSR 118.

- *FCOP and PIM*: Provide access to the native file system and personal information manager. The FCOP must provide properties giving the default URL of the storage directory for camera-captured photos and videos, wallpaper images, ring tones, music, and voice recorders, and a private storage directory for each MIDlet. PIM must include support for the `PIMList` subclasses `ContactList`, `ToDoList`, and `EventList` in read-only, read-write, and write-only modes. `ContactList` must support fax, home, mobile, preferred, and work numbers, as well as street-address, e-mail, note, URL, and photo fields. `ToDoList` must include the summary, priority, completion date, due, and completed fields. As with other portions of MSA Subset 1.0, these apply to MSA 1.0 as well.

- *Bluetooth*: If the device supports Bluetooth technology, it must include the Bluetooth API based on the GCF that JSR 82 describes. With this, applications can exchange media between devices using Bluetooth.

- *MMAPI*: The MMAPI must be included to permit developers to create compelling multimedia applications. Implementations must support the same resolutions for image capture from the image sensor as the system camera application.

- *Mobile 3D graphics*: Mobile 3D graphics are an important element for games, some rich GUIs, and other graphics-intensive applications. A compliant device must provide the interface that JSR 184 specifies.

- *WMA*: A compliant device must provide WMA 2.0, as messaging is a key phone feature that must be available to Java ME developers (see Chapter 14).

- *SVG*: Graphics remain a key component of today's and tomorrow's mobile applications; including support for JSR 226 ensures that applications can display SVG images.

Examining MSA 1.0

MSA 1.0 builds on MSA Subset 1.0 by adding the following packages:

- *J2ME Web services*: The J2ME Web Services Specification that JSR 172 describes—both the XML parser and the web services optional package—must be included to facilitate a standard way to support web services (see Chapter 13).

- *Security and trust services*: SATSA is an important element in a wide variety of mobile applications and services. SATSA-CRYPTO must be provided, while support for the APDU interface within SATSA and SATSA-PKI are conditionally mandatory (see Chapter 15).

- *Location*: The Location API that JSR 179 describes must be included if the device has a GPS receiver or other position-location acquisitions subsystem to enable location-based applications.

- *SIP*: SIP is the protocol of choice for IP-based telephony networks to establish and manage voice call handling. MSA 1.0 requires that a compliant device includes the package that enhances GCF support by providing SIP as JSR 180 specifies is required.

- *Content handling*: The Content Handler API that JSR 211 defines permits one Java application (or MIDlet) to launch another based on the content types it supports.

- *Advanced multimedia*: The Advanced Multimedia Supplements (AMMS) API, which I mention briefly in Chapter 16, must be included to permit developers to use enhanced camera support, 3D audio, support for an internal radio, and superior image encoding and processing capabilities.

- *Internationalization*: MIDlets and applications for Java ME are hard to internationalize; there's no good way to isolate localized application resources from the source code or dynamically identify those resources at runtime. JSR 238 provides a standard means for accomplishing this, and must be included.

Evolving for the Future: MSA2

JSR 249 defines MSA2, a collection of requirements that is backward compatible with MSA 1.0. Written by an expert group with substantially the same composition as JSR 248, MSA2 addresses changes in the Java ME marketplace that have occurred between the time JSR 248 development started and ended. It defines three platforms:

- *MSA2 Limited*: For low-tier, mass-market phones.

- *MSA2 Subset*: Targeted for mid-tier, mass-market phones, this is a superset of MSA2 Limited with additional requirements.

- *MSA2*: Targeted for high-tier devices, this is a superset of MSA2 Subset with additional requirements.

Like JSR 248, JSR 249 defines these platforms using the notion of mandatory requirements and conditionally mandatory requirements, and it not only specifies a collection of JSRs that each platform must support, but it also provides additional requirements about those JSRs, as well as additional requirements for the target platform as a whole. Some of these requirements are inherited from MSA 1.0 and from the JTWI; others are new to MSA2. Hardware requirements (such as screen and heap capabilities) remain the

same as for MSA 1.0; security requirements remain similar in principle but have expanded considerably to include privileges for the additional APIs that MSA2 requires.

■ **Caution** As I write this, JSR 249 is in draft form and is subject to change. Before you rely on the material I provide here, visit the JCP web site for this JSR at `http://jcp.org/en/jsr/detail?id=249` to obtain the latest copy and see if there are any changes that may affect your development plans.

Examining MSA2 Limited

MSA2 Limited is essentially MSA Subset 1.0, with only the minimum number of additions to ensure common APIs across the lowest tier of multiple devices. It includes the Mobile Internationalization API (JSR 238) that MSA 1.0 requires, as well as the generic framework for mobile device sensor management that JSR 256 defines.

The Mobile Sensor API gives you access to mandatory hardware sensors (the battery and network-received signal strength) and a 3D accelerometer if the device hardware has one.

Examining MSA2 Subset

MSA2 Subset builds on MSA2 Limited; in fact, it's a superset of MSA2 Limited. It must provide support for the following packages:

- *MIDP 3.0*: Defined in JSR 271, MIDP 3.0 is a fundamental part of MSA2 Subset and MSA2. Currently under development by members of the JCP, it will extend the MIDP 2.1 APIs already present by adding support for shared library code, better application life-cycle management, a secure binary interface for the inter-MIDlet exchange of record stores, and enhanced user interfaces, including required support for the MMAPI.

- *Mobile broadcast service*: Many devices today can receive broadcast content such as digital television. JSR 272 defines interfaces to access the service guide (permitting applications to search the database of program content) as well as render broadcast content including subtitles or other metadata associated with broadcast content using an API based on the MMAPI.

- *Contactless communication*: JSR 257 (which I mention in Chapter 15) describes means of performing contactless communication to RFID cards or visual tags. Support for this API is required if the underlying hardware provides these facilities.

- *Mobile user-interface customization*: Device personalization—wallpaper, ring tones, and device theming—is an important feature of many mobile devices today and a major source of revenue for content providers. JSR 258 defines an API that permits access to personalization data and provides a means for users to re-skin Java ME applications; this API is required.

- *XML*: XML is arguably the *lingua franca* of today's Web; JSR 280 defines standard interfaces that replace those that JSR 172 defines. JSR 280 enables better XML parsing and includes a streaming XML parser and support for the XML DOM. This API is required to enable application developers to work with web services.

Examining MSA2

Based on MSA2 Subset, MSA2 requires inclusion for the IP Multimedia Subsystem (IMS) services. IMS uses standard protocols including SIP, Diameter (a superset of RADIUS), MMS, and RTSP to enable IP-based real-time messaging, chat, push-to-talk (PTT), and multiplayer gaming that are rapidly becoming a standard part of the 3G wireless infrastructure worldwide. To make the most of the IMS backbone on these networks, downloaded applications must have access to IMS features through a standard API; JSR 281 defines this API.

JSR 281 includes support for many aspects of IMS in a Java-centric way, including definitions of optional packages to provide support for the following IMS features:

- *Push*: MIDlets can register to respond to incoming IMS connections, just as they can to other messages.

- *Quality of service (QoS)*: QoS guarantees are an important part of the IMS standard, and the IMS API permits streaming of audio and video with QoS guarantees.

- *Codecs*: IMS API implementations must include support for AMR-NB (and AMR-WB, the wideband version of AMR, if the device is a wideband device) as well as H.263 for streaming video if the device supports it.

The IMS API builds upon the conceptual frameworks that the GCF and the MMAPI provide.

Wrapping Up

Java ME's strength as a platform lies in its write-once, run-anywhere nature; unfortunately, this laudable goal is perpetually threatened by the open nature of the Java community, which encourages and embraces growth, leading to API and platform fragmentation. The Java community works to both extend the APIs available to Java ME as well as stem the fragmentation of the Java ME platform through the JCP, which issues JSRs that document additional APIs or combinations of APIs to enable new applications and services. Three key JSRs are working to reduce Java ME platform fragmentation by defining collections of Java ME APIs that should be included on as many devices as possible.

The first, JSR 185, defines the JTWI, a bare-bones mobile handset that supports the CLDC, the MIDP, and WMA. It may also support some audio multimedia through the MMAPI. Most of today's Java ME–enabled devices on the market explicitly comply with the JTWI, either as a deliberate, verified goal of the device manufacturer or as a matter of course.

The second two, JSRs 248 and 249, build on the JTWI and define additional platforms that require an increasing number of APIs. These JSRs permit you to rely on the presence of those required APIs for file system access, MMAPI, SATSA, Mobile 3D Graphics API, WMA, and SVG API, as well as APIs that implement new network features such as SIP and IMS.

APPENDIX

Finding Java APIs

Throughout this book, I've focused on specific required and optional Java ME APIs and how to use them. In some cases, I've pointed you to specific JSRs that document those APIs, while in other cases I haven't. Moreover, I've only discussed a subset of the optional Java ME APIs; many other optional APIs are available on some Java ME–enabled devices.

The plethora of JSRs about Java ME APIs is truly overwhelming, especially as you start out. Inevitably, someone will ask you, "Can a Java ME device do *X*?" Frequently, it's somebody important, such as the product manager designing your company's next product, which you'll spend the next six months of your life developing. Answering this question accurately isn't as easy as it sounds, especially if you're not familiar with *X* right away. You need to consult the JSRs relevant to Java ME, find the APIs that pertain to *X*, and see if they can do what you need. That's what this Appendix is for: it contains Table A-1, which maps important mobile technologies to the relevant JSRs that describe APIs for that technology. Armed with this information, you can go to `http://www.jcp.org`, look up the relevant JSR, and form the answer to the question you've been asked.

A final word: JSRs tell you *how* to do something with a Java ME device. They don't tell you if any Java ME devices *actually* do it that way. Mind you, I'm not knocking the JCP; it's a great process. I'm pointing out a reality of the free market around Java ME. The APIs that some JSRs describe are only implemented by a few device manufacturers, often because customers of those device manufacturers (either network operators or end consumers) don't understand the importance of the API set in the JSR. Other JSRs contain excellent ideas, but are simply too expensive to implement. And a few are quickly superseded or extended, so the JSR is only a fossil showing a snapshot of what happened at one moment in the evolution of an API. For these reasons, you should look at JSRs as only the first step in answering the question, "Can a Java ME device do *X*?" The information in a JSR lets you answer this question with the response, "Theoretically, yes," or "Not in a standard way" (because there may be proprietary ways to do *X* on some devices anyway). To provide a complete answer to the question—which probably really means, "How many Java ME devices can do *X*?" and "Do we have a viable business case to build our application that needs *X*?"—you also need to look at devices on the market in the time frame your product encompasses. That's more difficult in many cases; you need to take into account how your product will be distributed, whether or not it only pertains to customers of a single network operator, and so forth. Getting those answers takes legwork. Expect to spend time

searching the Internet, reading product data sheets and developer documentation for specific devices, and scouring developer forums about the devices in question to provide a clear answer as to the actual market availability of a specific feature or API.

Table A-1. *Java ME JSRs by Technology*

Technology	JSR	Notes
3D graphics	184, 239, 297	The MSA both requires and clarifies JSR 184.
Ad-hoc networking	259	
Advanced GUI	209	
Application deployment	232	This describes a mechanism for secure application deployment for CDC devices.
Bar-code recognition	257	The MSA both requires and clarifies JSR 257.
Bluetooth	82	This GCF-based implementation can provide both client and server end points.
Broadcast	272	The interfaces JSR 272 defines are based on the MMAPI and the GCF. The MSA clarifies JSR 272.
CLDC	30, 139	The JWTI and the MSA have clarifying remarks regarding CLDC implementations.
CDC	36, 218	
Contactless communication	257	The MSA both requires and clarifies JSR 257.
Content handling	211	The MSA both requires and clarifies JSR 211.
Cryptography	177	The MSA both requires and clarifies JSR 177.
Device sensors	256	The MSA both requires and clarifies JSR 256.
Digital set-top box	242	
File system	75	The MSA both requires and clarifies JSR 75.
Foundation Profile	46, 219	
Gaming	178	
IMS	281	The MSA both requires and clarifies JSR 281.
Internationalization	238	The MSA both requires and clarifies JSR 238.
Java database connectivity	169	
JTWI	185	
Location	179	The MSA both requires and clarifies JSR 179.
Messaging	120, 205, 266	JSR 205 introduced MMS. The JTWI both requires and clarifies 120. The MSA both requires and clarifies JSR 205.

Continued

Technology	JSR	Notes
MIDP	37, 118, 271	The JWTI and the MSA have clarifying remarks regarding MIDP implementations.
MMS	205, 266	JSR 205 introduced MMS. The MSA both requires and clarifies JSR 205.
Mobile broadcast	272	Interfaces are based on the MMAPI and the GCF. The MSA both requires and clarifies JSR 272.
MSA	248, 249	
Multimedia	135, 226, 234, 287	JSRs 226 and 287 describe SVG rendering.
Payments	229	
PDA optional packages for J2ME	75	The MSA both requires and clarifies JSR 75.
Personal Basis Profile	129, 217	
Personal Information Manager access	75	The MSA both requires and clarifies JSR 75.
Personal Profile	62, 216	
Remote Method Invocation	66	
RFID	257	The MSA both requires and clarifies JSR 257.
Security and trust	177	The MSA both requires and clarifies JSR 177.
Sensors	256	The MSA both requires and clarifies JSR 256.
SMS	120, 205, 266	The JWTI both requires and clarifies JSR 120. The MSA both requires and clarifies JSR 205.
SIP	180	JSR 180 defines a GCF-based implementation of classes that provide SIP support.
SVG	226, 287	The MSA both requires and clarifies JSR 226.
Telematics	298	
Telephony	253	
Television	272	This defines interfaces that are based on the MMAPI and the GCF. The MSA both requires and clarifies JSR 272.
Theming	258	The MSA both requires and clarifies JSR 258.
UI customization	258	The MSA both requires and clarifies JSR 258.
Visual tag recognition	257	The MSA both requires and clarifies JSR 257.
Web service APIs	172	The MSA both requires and clarifies JSR 172.
Wireless messaging (SMS, MMS)	120, 205, 266	JSR 205 introduced the MMS. The JTWI both requires and clarifies JSR 120. The MSA both requires and clarifies JSR 205.
XML	172, 280	JSR 280 supersedes JSR 172. The MSA both requires and clarifies both JSR 172 and JSR 280.

Index

Numbers

2D games, 193–218, 540
3D graphics
  JSRs and, 540
  MSA requirements and, 533
911 emergency systems, location-based services and, 499

Symbols

< > indicating plain text XML elements, 339
< /> indicating empty elements, in XML, 339
< less than, 339
<!-- --> indicating comments in XML documents, 339
</ > indicating starting tag, in XML, 339
<? ?> indicating preamble to XML documents, 339
> more than, 339
& ampersand, 339
/ solidus character, 163, 165
‘ ’ single quote, 339

A

A-GPS (Assisted Global Positioning System), 499
AAC format, 448
Abstract Windowing Toolkit. *See* AWT
acceptAndOpen method, 303
access modes, 297
AccessException, 245
accessor field keys, 177
ActionEvent class, 266
ActionListener, 265
actionPerformed method, 266
actions, for WeatherApplet sample application, 61–75
Activatable class, 278
active state
  of Xlets, 224
  of MIDlets, 85
ad-hoc networking, JSRs and, 540
add methods (Container class), 262
addAddress method, 383, 384
addBinary method, 182
addBoolean method, 183
addCategory method, 184, 507
addCommand method, 106, 195
addContact method, 189
addDate method, 182
addFileSystemListener method, 168
addInt method, 182
addLandmark method, 508
addLocation method, 154, 160
addMessagePart method, 383
addPlayerListener method, 460
addProximityListener method, 507
addRecord method, 138, 141
addRecordListener method, 145
addresses, MMS messages and, 383
AddressInfo class, 502
addStore method, 140
addString method, 183
addStringArray method, 183
addTargetListener method, 435, 439
Adobe
  Illustrator, 471
  Shockwave Flash, 470
Advanced Graphics and User Interface. *See* AGUI
AES (Encryption Standard), 423
AGUI (Advanced Graphics and User Interface) 230
  developing user interfaces with, 266–271
  limitations of, 269–271

AIDs (application identifiers), 417, 421
alarmFired method, 95
alarms, 90–96
Alert class, 99, 114, 116
AlertType enumeration, 116
AlreadyBoundException, 245
ampersand (&), 339
AMS (Application Management Software)
  JAR/JAD files and, 89
  MIDlets and, 86
animated SVG images, 477
animations, 205–207
Ant (Apache), 36
APDU (Application Protocol Data Unit), 417
APDUConnection interface, 418
API permissions, 54
append method, 106, 114, 202
appendChild method, 483
<APPLET>, 258
Applet Descriptor section, 54
applet model, 14
AppletContext class, 257
applets, 253–260
  communicating between, 258
  life cycle of, 254
  vs. MIDlets/Xlets, 254, 258
  naming, 258
application identifiers (AIDs), 417, 421
Application Management Software (AMS)
  JAR/JAD files and, 89
  MIDlets and, 86
Application Protocol Data Unit (APDU), 417
application screens, for user interfaces. *See* screens
applications
  adding location information to, 509–520
  building/running, 43–53
  deploying, JSRs and, 540
  distributed object-oriented, 273
  generating XML in, 343–355
  marketing/selling yours, 17
  packaging/executing, 53–56
  samples of. *See* sample applications; source code
  security for, 413–445
  signing, 55
  steps for using web services from, 341
ARGS property, 242
ARPU (average revenue per user), 3, 4
Assisted Global Positioning System (AGPS), 499
asymmetric cryptography, 423
AsymmetricBlockCipher interface, 429
atomicity, record stores and, 135, 142
average revenue per user (ARPU), 3, 4
AWT (Abstract Windowing Toolkit)
  developing user interfaces with, 260–266
  Personal Basis Profile and, 14
  Personal Profile and, 15
  user-interface model and, 99
AWT class hierarchy, for user-interface components, 261
azimuthTo method, 506

B

BadPaddingException, 425
bar codes. *See* visual tags
BD-J stack, 231
Berners-Lee, Tim, 306
BinaryMessage interface, 376, 379, 381
bitmap images, 470
block ciphers, 425
BlockCipher interface, 429
blocks of notes, 468
Blu-ray Disc Java (BD-J), 231
Bluetooth
  JSRs and, 540
  MSA requirements and, 533
–bootpathoption, 52
BorderLayout manager, 262
Bouncy Castle API, 425–431
  message digests created via, 428
  vs. SATSA-CRYPTO API, 430
broadcast service
  JSRs and, 540
  MSA2 and, 535
build directory, 36
build.xml file, 36

butterfly/cat game (sample application), 207–217
Button class, 264
buttons, 236–240
ByteArrayInputStream class, 95, 139, 140, 142, 362
ByteArrayOutputStream class, 95, 139, 140

C

calendar appointments, 161, 174
  creating, 183
  PIMItem class and, 177
  removing, 184
Calendar class, 28
callSerially method, 396
cameras
  capturing media and, 462
  imaging sensors and, 449, 461, 464
  MMAPI support and, 464
cancel method, 318
canRead method, 166
Canvas class, 99, 122, 193, 460, 463
  vs. CustomItem class, 125
  in Java ME vs. Java SE, 122
  NetBeans IDE Mobility Pack and, 483
  rendering SVG images and, 477
  vs. Screen class, 114
canvas, 122–131
canWrite method, 166
capture method, 463
capture scheme, 455
capturing media, 461–466, 494
CardLayout manager, 262
case sensitivity, XML and, 339
cat/butterfly game (sample application), 207–217
categories, PIM database and, 176, 184
CDC (Connected Device Configuration), 8, 12
  AGUI Optional Package and, 266
  GUI toolkits and, 230
  history of, 23
  JSRs and, 540
CDC applications. *See* Xlets
CDMA networks, 378
cell phones. *See* mobile phones
cells, 201
certificates, 55, 415
certification, by third parties, 17
changePin method, 420
character entities, 339
characters event, 357, 360
child elements, XML and, 336
Choice class, 112
ChoiceGroup class, 112
choices
  displaying, 120
  managing, 112
Cipher interface, 424
ciphers, 415
class loaders, 242
CLDC (Connected Limited Device Configuration), 8, 10
  Connection implementations and, 299
  Contactless Communication API packages and, 432
  GCF and, 293
  history/future of, 19–22
  JSRs and, 540
  JTWI requirements and, 529
  MSA requirements and, 533
CLDC 1.0, 22
CLDC 1.1, 19
CLDC/MIDP applications. *See* MIDlets
close method, 173, 295
closed state, 312, 453
closeRecordStore method, 137
closeStore method, 153
code division multiple access (CDMA) networks, 378
code samples. *See* source code
code signing, 190
codecs, 448–453
Collections API (Java SE), 28
collidesWith method, 207, 217
collision detection, 207, 217
Command class, 101
commandAction method, 101, 122, 325, 491, 493

CommandListener class, 101, 122
commandListener method, 493
commands, 101, 106, 123
comments
  NetBeans IDE and, 50
  in XML documents, 339
commit method, 183
compare method, 143
Component class, 234, 261, 263
components, 234, 260–266, 270
Concurrent Versions System (CVS), 36
configurations, 8, 10
configViewSound method, 492, 494
configViewSvg method, 492, 496
configViewVideo method, 492, 494
CONNECT method, 307
Connected Device Configuration, *See* CDC
Connected Limited Device Configuration. *See* CLDC
connected state, HTTP communication and, 312
Connection class, 296–299, 378
Connection implementations (table), 299
Connection interface, 295
Connection schemes (table), 297
Connection subclasses, contactless connections and, 432, 435, 440
ConnectionNotFoundException, 165, 295, 296
Connector class, 162, 295
  datagram communication and, 304
  socket communication and, 300
Connector.READ access mode, 297
Connector.READ_WRITE access mode, 297
Connector.WRITE access mode, 297
consumer-producer model, 259
consumers, Java ME market perspective and, 3, 5
Contact class, 181
Contactless Communication API, 431
contactless communications, 431–444
  contactless targets and, 432, 435–439
  JSRs and, 540
ContactList class, 189, 533
ContactLoaderThread class (sample), 189
contacts, 161, 174, 189
  creating, 183
  PIMItem class and, 177–182
  removing, 184
Container class, 234, 235, 256, 261
containers, 234, 256
  AWT, 261–263
  multimedia content and, 448
content
  JSRs and, 540
  multimedia. *See* multimedia content
  remote, 309
content bounds, 125
content ID, MMS messages and, 383
ContentConnection class, 295, 309–315
  vs. HttpConnection class, 311–315
  StreamConnection interface and, 311
ContentDescriptor class, 451
Control interface, 450, 451
control loop, 195, 197
Control subclasses, 458, 494
controllers, multimedia content and, 449
cookies, 335
Coordinates class, 502, 506
countermeasures, security and, 414, 415
createAnimatedTile method, 205
createBoard (sample) method, 216, 217
createButterflies (sample) method, 216, 217
createCat (sample) method, 216, 217
createContact method, 183, 189
createElementNS method, 483
createEmptyImage method, 482
createEvent method, 183
createLandmarkStore method, 507
createPlayer methods, 454
createToDo method, 183
creating
  calendar appointments, 183
  contacts, 183
  custom items, 125
  directories, 165
  events, 183
  files, 165

lightweight components, 236–240
message digests, 422, 428
messages, 379
MIDlets, 37–56
PIM records, 183
to-do items, 183
user interfaces, 97–131
WeatherWidget sample application, 38–51
Xlets, 57–77, 225, 227–233
Criteria class, 502, 504
cryptographic signatures, 17
cryptography, 414–425
asymmetric, 423
JSRs and, 540
CustomItem class, 107, 122, 125–131
CVS (Concurrent Versions System), 36

D

daemon threads, 27
data communication, 293–329
HTTP communication and, 306–326
permissions and, 327
socket/datagram communications and, 300–306
Data Encryption Standard (DES), 423
data representation, XML for, 336–341
data services, 4
data sharing, Xlets and, 243–251
data sources, multimedia content and, 449
database connectivity, JSRs and, 540
database of landmarks, location-based services and, 501
Datagram class, 304–306
datagram communication, 300, 304–306
Datagram interface, 295
DatagramConnection class, 304
DatagramConnection interface, 295
DataInput interface, 295, 305
DataInputStream class, 95, 139, 140, 142, 311
opening files and, 166
reading files and, 173
DataLengthException, 430
DataOutput interface, 295, 305
DataOutputStream class, 95, 139
reading files and, 173
sockets and, 301
DataSource class, 450, 451, 455
date and time, 110, 138
Date class, 28
date-book appointments, 161, 174
creating, 183
PIMItem class and, 177
removing, 184
DateField class, 110
deallocate method, 453
decode method, 444
decrypting messages
Bouncy Castle API and, 429
SATSA-CRYPTO API and, 423
DefaultHandler class, 357
defense in depth, 415
defineReferencePixel method, 206
DEFRecordListener interface, 433
DELETE method, 307, 334
delete method
Choice class, 114
FileConnection class, 167
TextField class, 110
deleteAll method, 114
deleteCategory method, 184, 507
deleteLandmark method, 508
deleteLandmarkStore method, 507
deleteRecord method, 138, 144
deleteRecordStore method, 137
DES (Data Encryption Standard), 423
deserialization, 140, 150
destroy method, 254
destroyApp method, 85, 87, 325, 491
destroyed state
of MIDlets, 85
of Xlets, 225
destroyXlet method, 225, 233
determineLocation method, 518
development planning, 16–18
device manufacturers, Java ME market perspective and, 3

device sensors, JSRs and, 540
DeviceKeyEvent class, 270
devices
  code signing and, 190
  development planning, 16
  FCOP and, 162
  file system changes, listening for, 168
  JTWI requirements/options and, 529
  location-based services for, 499–521
  MMAPI supplemental interfaces and, 464
  MMAPI system property definitions and, 457
  MSA mandatory components/requirements and, 531
  personalizing, 536
  resolving fragmentation and, 523–537
  socket/datagram communications and, 300
  supported media/protocol types, enumerating on, 457
Digest interface, 429
digital signal processors (DSPs), multimedia content and, 448
digital signatures, 415
directories
  creating, 165
  deleting, 167
directorySize method, 166
disablePIN method, 420
DiscoveryManager class, 432, 433, 440
display canvas, 122–131
Display class, 98, 483
Displayable class, 99, 107
  GameCanvas class and, 195, 199
  sample game application and, 209
Displayable interface, 85
dist directory, 36
distance method, 506
distributed object-oriented applications, 273
distribution considerations, 17
Document class, 474, 483
Document Type Definition (DTD), 341
doFinal method, 425
DOM parsers, 342
double buffering, 124
downloads
  Bouncy Castle API, 426
  kXML parser, 365
  Mobility Packs, 34
  NetBeans IDE, 34
drawing
  Canvas class and, 123
  controlling via Canvas class, 122–125
  GameCanvas class and, 195–200
  Item class and, 106
DSPs (digital signal processors), multimedia content and, 448
DTD (Document Type Definition), 341
dynamic registration for inbound messages, 390

E

EAN/UPC (European Article Number/Universal Product Code), 440
EclipseME, 34, 53
Element interface, 474
empty elements in XML, indicated by < />, 339
enablePIN method, 420
encode method, 444
encoding MMS messages and, 383
encrypting messages
  Bouncy Castle API and, 429
  SATSA-CRYPTO API and, 423
Encryption Standard (AES), 423
endDocument event, 357, 361
endElement event, 357, 360
enterPIN method, 420
enumerateRecords method, 142
error handling, wireless messages and, 398
European Article Number/Universal Product Code (EAN/UPC), 440

event handling
Canvas class and, 123
GameCanvas class and, 195–200
Item class and, 106
Event interface, 474
EventList class, 533
EventListener, 474
events, 161, 174, 264
creating, 183
PIMItem class and, 177
PlayerListener interface and, 460
removing, 184
exceptions
addFileSystemListener method and, 169
addRecord method and, 141
creating interfaces and, 165
data communications and, 295
deleteRecord method and, 144
deleteRecordStore method and, 137
encrypting/decrypting messages and, 425
getNextRecordID method and, 142
getNumRecords method and, 145
getRecord method and, 142
items method and, 177
listRoots method and, 169
location-based services and, 503
message digests, 423
open method and, 165, 296
openRecordStore method and, 136
PIM package and, 175
removeFileSystemListener method and, 169
setRecord method and, 144
verifyPIMSupport method and, 189
visual tags and, 444
exchangeAPDU method, 419
exchangeData method, 436
executing
MIDlets, 53–56
Xlets, 75
exit method, 27
exitMIDlet method, 95, 325
exportObject method, 284
ExtendedDigest interface, 429
extensibility, 7, 15
ExternalResourceHandler, 475, 482, 496

F

FCOP (File Connection Optional Package), 161–173
determining if present, 164
Java ME fragmentation and, 524
LocationStore sample class and, 169–173
MSA requirements and, 533
field keys, PIM package and, 177
field types, 141
file access, 161–191
FCOP and, 161–173
PIM package and, 174–190
File Connection Optional Package. *See* FCOP
file:/// protocol prefix, 163
file systems
adding/removing, 168
FCOP and, 161–173
JSRs and, 540
listening for changes to, 168
FileConnection class, 162–173
integrating into WeatherWidget, 173
vs. RecordStore class, 173
files, working with, 165–167
fileSize method, 166
FileSystemRegistry class, 168, 169
finalize method, 26
flat character data, XML and, 339
floating-point mathematics, CLDC and, 12, 26
Flow Designer, 38–50
FlowLayout manager, 262
focus events, 126
FocusListener interface, 265
Form class, 99
collecting visible items via, 114
items, adding to, 104

FP (Foundation Profile), 12, 14, 231
  AGUI Optional Package and, 266
  JSRs and, 540
fragmentation, Java ME and, 523–537
Frame class, 234
Frame container, 262
FramePositioningControl interface, 458
frames, 201
freeMemory method, 27
fromBytes method, 150, 355
fromXml method, 361, 372

G

GameCanvas class, 99, 122, 194, 195–200
  Canvas class and, 195
  SpriteCanvas class and, 210–217
games, 193–218
  cat/butterfly sample game and, 207–217
  JSRs and, 540
  managing execution for, 197
garbage collection, gc method for, 27
Gauge class, 111
gc method, 27
GCF (Generic Connection Framework), 11, 163, 293–329
  APDU hierarchy and, 417
  contactless targets and, 435
  datagram communication and, 304–306
  hierarchy of, 295
  HTTP communication and, 309–315
  permissions and, 327
  socket communication and, 300–303
  using, 295
generateVisualTag method, 444
Generic Connection Framework. *See* GCF
GET method, 307, 311, 313, 319
getAddress method, 301, 380
getAddressInfo method, 505, 508
getAlgorithmName method, 429
getAltitude method, 506
getApplet method, 257
getAppletContext method, 257
getApplets method, 257
getAppProperty method, 87, 89
getAttributeValue method, 371
getAudioClip method, 257
getBinary method, 179
getBlockSize method, 430
getBoolean method, 180
getByteLength, 429
getCaretPosition method, 110
getCategories method, 184, 507
getChars method, 110
getCipherSuite method, 326
getClassLoader method, 226
getComponent method, 263
getComponentAt method, 263
getConditions method, 131
getContainer method, 226
getContent method, 383
getContentAsStream method, 383
getContentID method, 383
getContentLocation method, 383
getContentType method, 453
getControl method, 453, 458
getControls method, 453
getCourse method, 505
getDate method, 110, 179, 312
getDescription method, 508
getDeviceKeyCode method, 270
getDigestSize method, 429
getDisplay method, 98
getEncoding method, 311, 383
getExpiration method, 312
getExtraInfo method, 505
getFile method, 313
getFilter method, 391
getFont method, 108
getForecast method, 281
getGraphics method, 124
getHeader method, 383, 385
getHeaderField method, 312
getHeaderFieldKey method, 312
getHorizontalAccuracy method, 506
getHost method, 313
getImage method, 118, 257
getImageProperties method, 444
getIndicator method, 118
getInitialReference method, 421

getInputBlockSize method, 430
getInputMode method, 110
getInstance method, 397, 423, 504, 507
getInt method, 180
getInteractionModes method, 125
getKeyStates method, 196
getLabel method, 107
getLandmarks methods, 508
getLastKnownLocation method, 505
getLastModified method, 138, 312
getLatitude method, 506
getLayerAt method, 202
getLength method, 311, 314
getLocalAddress method, 301
getLocalPort method, 301
getLocation method, 154, 263, 281, 505, 506
getLocationMethod method, 505
getLocationOnScreen method, 263
getLocationStrings method, 153, 160
getLongitude method, 506
getMessage method, 403
getMessageParts method, 383
getMidlet method, 391
getMIMEType method, 383
getMin method, 477
getMinContentHeight method, 125, 477
getMinContentWidth method, 125, 477
getMinimumHeight method, 107
getMinimumSize method, 240
getMinimumWidth method, 107
getName method, 166, 508
getNextRecordID method, 142
getNumRecords method, 138, 145
getOutputBlockSize method, 430
getPath method, 166
getPayloadData method, 381
getPayloadText method, 380
getPort method, 301, 313, 326
getPref method, 477
getPrefContentHeight method, 125, 131, 477
getPrefContentWidth method, 125, 131, 477
getPreferredHeight method, 107
getPreferredSize method, 240
getPreferredWidth method, 107
getProperty method, 27, 242, 406, 419, 457, 463, 464
getProtocol method, 313
getProtocolName method, 326
getProtocolVersion method, 326
getQualifiedCoordinates method, 505, 508
getReadSymbologies method, 440
getReason method, 175
getRecord method, 142
getRecordType method, 437
getResource method, 242
getResourceAsStream method, 242, 469
getResponseCode method, 312, 314
getResponseMessage method, 312
getRuntime method, 27
getSecurityInfo method, 326
getSelectedFlags method, 114
getSelectedIndex method, 114
getServerCertificate method, 326
getSize method, 138, 202, 263
getSizeAvailable method, caution with, 146
getSnapshot method, 461–465, 494
getSocketOption method, 301
getSpeed method, 505
getStartContentId method, 382
getState method, 460
getString method, 110, 118, 180
getStringArray method, 180
getSubject method, 383
getSupportedContentTypes method, 457, 492
getSupportedFields method, 179
getSupportedProtocols method, 457
getSupportedTargetTypes method, 435
getSystemResource method, 242
getSystemResourceAsStream method, 242
getTargetComponent method, 478
getText method, 371
getTimeBase method, 453
getTimeout method, 118
getTimestamp method, 380, 382, 505
getType method, 311, 360
getURL method, 167, 313
getValue method, 360

getVerticalAccuracy method, 506
getXletProperty method, 226, 242
get_alarmAlert method, 96
get_exitCommand method, 96
get_helloStringItem method, 95
get_infoForm method, 95
get_listContacts method, 189
get_mainForm method, 105
Global Positioning System (GPS), 500
Global System for Mobile (GSM), 374, 378
GPS (Global Positioning System), 500
Graphics class, 122, 125, 235, 257
graphics operations, 124
graphics. *See* images
GridBagLayout manager, 262
GridLayout manager, 262
GSM (Global System for Mobile), 374, 378
GUIControl interface, 458, 459
GUIs, JSRs and, 540

H

H.263 video streams, 448
hasPointerEvents method, 123
hasPointerMotionEvents method, 123
HAVi (Home Audio Visual Interoperability), 231
HEAD method, 307
headers
  HTTP communication and, 308
  MMS messages and, 383, 385
heap
  JTWI recommendations for, 528
  MSA requirements for, 532
heavyweight components, 264, 270
"Hello World!"
  datagram communication and, 304
  HTTP communication and, 307
  wireless messages and, 381
Hello World MIDlet, 83
"Hello Xlet", 229, 232
HelloMidlet method, 85
high-level events, 264
Home Audio Visual Interoperability (HAVi), 231
HTTP communication, 298, 306–326, 334
  GCF and, 309–315
  methods for, 307
  opening connections and, 295
  securing via HTTPS, 325
  socket/datagram communications and, 300
  status codes for, 308
  WeatherWidget sample application and, 315–325
HTTP methods, 334
http scheme, 455
HttpConnection class, 311–315
HTTPS communication, 299, 325, 334
  certificates and, 415
  vs. SATSA-CRYPTO API , 423
HttpsConnection class, 326

I

IllegalArgumentException, 136
  open method and, 165, 296
  playTone method and, 466
IllegalBlockSizeException, 425
IllegalStateException
  ciphers and, 425, 430
  Player instances and, 453, 463
Illustrator (Adobe), 471
ImageItem class, 111
images
  sending in messages, 384, 398–407
  visual tag recognition and, 432
imaging sensors, 449, 461, 464
IMS (IP Multimedia Subsystem)
  JSRs and, 540
  MSA2 and, 536
indexes, for records, 135, 138
init method, 254, 491
initComponents method, 263
initDisplayMode method, 459, 460, 463
initialize method, 95, 159, 325
initMediaPlayer method, 492, 493
initSvgPlayer method, 492, 496
initXlet method, 225, 233
Inkscape, 471

input constraints, 109
input devices, AGUI and, 269
input modes, 109
InputConnection interface, 295
InputStream class, 295, 301, 315
InputStreamReader class, 30
insert method, 106, 202
  Choice class, 114
  TextField class, 110
integrated development environments, 33, 53
Inter-Xlet Communication (IXC), 243
International Organization for Standardization (ISO), 417
internationalization, MSA and, 534, 540
interrupt method, 27
invalidate method, 239
InvalidKeyException, 425
InvalidRecordIDException, 142
invokeAndWait method, 269, 479
invokeLater method, 269, 479, 483
IOException, 165, 295, 296, 444
IP Multimedia Subsystem (IMS), MSA2 and, 536
isAncestorOf method, 263
isCategory method, 184
isDirectory method, 167
isDoubleBuffered method, 124, 270
isHidden method, 167
ISO (International Organization for Standardization), 417
ISO14443Connection class, 436
isOpen method, 167
isSelected method, 114
isSupportedField method, 179
isValid method, 505
Item class, 99, 106
items, 104–114
  custom, 122–131
  PIM database categories and, 184
items method, 177
itemsByCategory method, 184
itemStateChanged method, 116
IXC (Inter-Xlet Communication), 243, 246
IxcRegistry, 246

J

J2ME Polish, 17
J2ME Web Services Specification, 331, 355–365
JAD files, 6
  JTWI devices and, 528
  MIDlets and, 53–56, 87, 89, 387–390, 532
JAD/JAR pairs (suites), 53
JAR files
  MIDlets and, 53, 87, 89
  Xlets and, 75, 242
Java API for XML-based RPC (JAX-RPC), 355
Java APIs, 539–541
Java Application Descriptor. *See* JAD files
Java AWT, user-interface model and, 99
Java Community Process (JCP), 523–528, 530, 535–537
Java database connectivity, JSRs and, 540
Java ME (Java Platform, Micro Edition)
  architecture of, 10–16
  development planning and, 16–18
  fragmentation and, 523–537
  market for, 3
  marketing/selling your applications and, 17
  platforms/libraries comprising, 6
  security and, 6, 413–445
  XML support for web services and, 341–372
Java ME Device Table, 300
Java ME Scalable 2D Vector Graphics. *See* SVGAPI
Java Mobile Game API, 193–218, 540
Java Network Launching Protocol (JNLP), 75
Java Platform, Micro Edition. *See* Java ME
Java Remote Method Protocol (JRMP), 275
Java RMI stack. *See* RMI stack
Java SE (Java Standard Edition)
  vs. CDC, 23
  vs. CLDC 1.1, 20
  Collections API of, 28
  generating stub classes and, 284
  implementing remote services and, 283
  RMI OP and, 279
  rmic compiler and, 281

Java SE class library
CDC and, 31
CLDC and, 24–30
Java SE Java Development Kit, 34
Java smart cards, 416–421
Java Specification Requests. *See* JSRs
Java Standard Edition. *See* Java SE
Java streams, 139
Java Technology for the Wireless Industry (JTWI), 528, 540
Java Verified Program, 17, 415
Java Virtual Machine (JVM), 10
Java Web Start, 75
java.io package, changes in to fit CLDC, 29
java.lang package, changes in to fit CLDC, 24–28
java.util package, changes in to fit CLDC, 28
JavaCardRMIConnection interface, 418, 421
JavaScript Object Notation (JSON), 331
JAX-RPC (Java API for XML-based RPC), 355
JComponent class, 270
JCP (Java Community Process), 523–528, 530, 535–537
JCRMI interface, 420
JFrame class, 269
JMenuBar class, 270
JNLP (Java Network Launching Protocol), 75
JOptionPane class, 269
JPopupMenu class, 269
JRMP (Java Remote Method Protocol), 275
JSON (JavaScript Object Notation), 331
JSRs (Java Specification Requests), 8, 539–541
AGUI and, 230
CDC-enabled devices, user-interface packages and, 230
how to read, 525
Java ME fragmentation and, 523–528
JSR 30 - J2ME Connected, Limited Device Configuration, 529
JSR 36 - Connected Device Configuration, 23
JSR 63 - Java API for XML Processing 1.1, 356
JSR 62 - Personal Profile Specification, 253
JSR 66 - RMI Optional Package Specification Version 1.0, 278
JSR 75 - PDA Optional Packages for the J2ME Platform, 161, 174, 190
JSR 82 - Java APIs for Bluetooth, 533
JSR 118 - Mobile Information Device Profile 2.0, 533
JSR 120 - Wireless Messaging API, 373, 375, 529
JSR 135 - Mobile Media API, 447, 451, 465
JSR 139 - Connected Limited Device Configuration 1.1, 529, 533
JSR 172 - J2ME Web Services Specification, 331, 355
JSR 177 - Security and Trust Services API for J2ME, 416
JSR 179 - Location API for J2ME, 499
JSR 184 - Mobile 3D Graphics API for J2ME, 533
JSR 185 - Java Technology for the Wireless Industry, 524, 528–529
JSR 197 - Generic Connection Framework (GCF) Optional Package for J2SE, 294
JSR 205 - Wireless Messaging API 2.0, 373, 375, 386
JSR 206 - Java API for XML Processing (JAXP) 1.3, 386
JSR 209 - AGUI Optional Package, 266
JSR 216 - Personal Profile 1.1, 253
JSR 218 - Connected Device Configuration 1.1, 23
JSR 226 - Scalable 2D Vector Graphics API for J2ME, 472, 533

JSR 234 - Advanced Multimedia Supplements, 464
JSR 238 - Mobile Internationalization API, 534–535
JSR 248 - Mobile Service Architecture, 524, 530
JSR 249 - Mobile Service Architecture 2, 524, 530, 534
JSR 256 - Mobile Sensor API, 535
JSR 257 - Contactless Communication API, 431, 535
JSR 258 - Mobile User Interface Customization API, 536
JSR 271 - Mobile Information Device Profile 3, 535
JSR 272 - Mobile Broadcast Service API for Handheld Terminals, 535
JSR 281 - IMS Services API, 536
JSR 287 - Scalable 2D Vector Graphics API 2.0 for Java ME, 447, 470, 472, 480
life cycle of, 526
JTextComponent class, 269
JTWI (Java Technology for the Wireless Industry), 528, 540
JVM (Java Virtual Machine), 10

K

K virtual machine (KVM), 10
key codes, 123, 196
key events, 270
key generation, 430
key states, polling for, 195, 196
KeyEvent class, 270
KeyListener, 265
keyPressed method, 123, 126, 483
keyReleased method, 123, 126
keyRepeated method, 123, 126
keys, ciphers and, 415
keystrokes, polling for, 196
KVM (K virtual machine), 10
kXML parser, 365–372, 509
KXmlParser class, 365, 371

L

labels, 101
Landmark class, 502, 507
LandmarkException, 503
landmarks
location-based services and, 501
managing, 507
LandmarkStore class, 502, 507
lastModified method, 167
Layer class, 194, 200
LayerManager class, 194, 200–205
method of, 202
sample game application and, 216
layers, 200–205
layout flags, 107
layout managers, 261, 262
layout preferences, Item class and, 106
LBS. *See* location-based services
less than (<), 339
lightweight components, 233–241, 264, 270
List class, 99, 113
complex user interfaces and, 114
displaying choices via, 120
list method, 167
listConnections method, 303, 389
listeners, 265
listLandmarkStores method, 507
listRecordStores method, 137
listRoots method, 169
load method, 173
Location API, 499, 501–521
determining device location via, 503–507
JSRs and, 540
managing landmarks and, 507
permissions and, 508
Sun Java Wireless Toolkit and, 518
WeatherWidget sample application and, 509–520

Location class, 154, 502, 504, 517
  implementing, 146–150
  integrating into WeatherWidget, 155–160
  mutators and accessors and, 246
  RMI OP and, 281
  XML and, 343–355
location-based services (LBS), 499–521
  determining device location and, 503–507
  landmarks and, 501, 507
  security and, 508
  WeatherWidget sample application and, 509–520
Location interface, 282
LocationException, 503, 505
LocationImpl class (sample), 281, 283, 284
LocationListener interface, 502, 506
LocationParser class, 367–372, 517
LocationParserHandler class, 357–361, 370
LocationProvider class, 502–507
LocationStore class, 146, 169–173, 325, 507, 517
  implementing, 150–155
  integrating into WeatherWidget, 155–160
locationUpdated method, 502
LocationUpdater class, 518
locators, multimedia content and, 454, 463, 493
low-level events, 264

M

Mac OS X, NetBeans Mobility Pack and, 34
Manager class, 451, 454
manifests, 87
marketing considerations, 17
MarshalledObject class, 277
marshalling data, 342, 355
match method, 143
Math class, 26
media player (sample application), 484–496
media types, multimedia content and, 448
MediaException class, 451, 463
memory, 27
menu priorities, 270
menu types, 270
Message class, 376, 380
message counters, 403
message digests, 335, 415
  Bouncy Castle API and, 428
  SATSA-CRYPTO API and, 422
MessageConnection interface, 376–380, 385
MessageDigest class, 423
MessageListener interface, 376, 385
MessagePart class, 376
  vs. MultipartMessage class, 380, 383
  sendMMS method and, 406
messages. *See* wireless messaging
MetaDataControl interface, 458
microphones, 465
MIDIControl interface, 458
MIDlet suites, 133–137
MIDlets, 13, 83–96. *See also* WeatherWidget
  vs. applets, 254, 258
  attributes for, 87
  building/running, 43–53
  creating, 37–56
  executing, 53–56
  exiting, 87
  GameCanvas class, tying to, 199
  Hello World example of, 83
  incoming messages, registering dynamically for, 390
  JAD/JAR pairs and, 53
  life cycle of, 85, 97
  methods for, 85
  Mobility Pack for, 34, 37
  MultimediaMIDlet class and, 486–492
  packaging, 53–56, 87
  properties for, 89
  record stores and, 133
  resources and, 89
  sample game application and, 209
  startup events/alarms and, 90–96
  states of, 85

MIDletStateChangeException, 87
MIDP (Mobile Information Device Profile), 9, 13
  2D games and, 193
  Connection implementations and, 299
  HTTPS and, 325
  JSRs and, 541
  JTWI requirements and, 529
  MIDlets and, 83–96
  MIDP 3.0 and, 535
  MSA requirements and, 533
MIME type, MMS messages and, 383
MMAPI (Mobile Media API), 407, 448–469
  classes/interfaces of, 451
  JTWI requirements and, 529
  MMAPI system property definitions and, 457
  MSA requirements and, 533
  organization of, 450
  sample media player and, 484–496
  supplemental interfaces for, 464
  using, 452–469
MMAPI schemes, 454
MMS Center, 375
MMS messages, 374
  contents of, 383
  JSRs and, 541
  parts and, 376, 380, 382
  sending/receiving, 398–407
MMSMIDlet class, 399–404
MMSSender class, 404, 494
MO (mobile-originated) messages, 374, 379
Mobile 3D graphics, MSA requirements and, 533
mobile broadcast service
  JSRs and, 541
  MSA2 and, 535
mobile devices
  development planning and, 16
  Java ME platform and, 6–10
  Moore's law and, 21
Mobile Information Device Profile. *See* MIDP
Mobile Media API. *See* MMAPI
mobile phones, 6
  location-based services and, 499
  MIDP and, 9
  MSA and, 530–536
Mobile Sensor API, 535
mobile sensors, JSRs and, 541
Mobile Service Architecture (MSA), 530–536, 541
Mobile Station International Subscriber Directory Number (MSISDN), 391
Mobile User Interface Customization API, 541
mobile-originated (MO) messages, 374, 379
mobile-terminated (MT) messages, 374, 385
Mobility Pack for NetBeans, 483, 485
Mobility Packs, for MIDlets and Xlets, 34, 37, 57
model-view-controller (MVC), 99
Moore's law, 21
more than (>), 339
MouseListener, 265
MouseMotionListener, 265
MouseWheelListener, 265
moveButterflies (sample) method, 217
moveCat (sample) method, 199, 217
MP3 players, 454
MPEG-4, 448
MSA (Mobile Service Architecture), 530–536, 541
MSA2 (Mobile Service Architecture 2), 534
msgAvail message counter, 403
MSISDN (Mobile Station International Subscriber Directory Number), 391
MT (mobile-terminated) messages, 374, 385
multimedia content, 447–498
  capturing, 461–466
  IMS and, 536
  JSRs and, 541
  MMAPI and, 448–469, 484–496
  packaging and delivery of, 448
  playing audio/video and, 493
  playing SVG content and, 496
  rendering, 452–469
  SVGAPI and, 470–484

Multimedia Messaging Service. *See* MMS messages
MultimediaMIDlet class, 486–492, 496
MultipartMessage interface, 376, 379
  managing multiple parts of, 382
  message headers and, 385
multithreading, 27
MVC pattern
  MIDlets and, 99
  multimedia content and, 449

N

namespaces, XML and, 340
Naming class, 277
naming conventions
  for applets, 258
  for record stores, 135
NanoXML parser, 372
nbproject directory, 36
NDEF (NFC Data Exchange Format), 431, 436
NDEF-enabled targets, registering listeners for, 433
NDEFTagConnection class, 436
near-field communications, 431
NetBeans, 105, 107
  adding/removing file systems and, 169
  containers and, 263
  GUI builder and, 517
  Mobility Pack for, 483
  PIM package and, 189
  RMI applications and, 282
  SVG images and, 483
NetBeans IDE, 33–77
  comments in, 50
  home page of, 34
  installing, 34
  MIDlets and, 37–56
  reasons for using, 33
  Xlets and, 57–77
newMessage method, 379, 384
newSAXParser method, 362
next method, 365
nextFrame method, 206
nextRecord method, 143
nextTag method, 365
nextText method, 365
nextToken method, 365
NFC (Nokia Near Field Communication) SDK, 431
NFC Data Exchange Format (NDEF), 431, 436
Nokia Near Field Communication (NFC) SDK, 431
NoSuchAlgorithmException, 423, 425
NoSuchMethod exception, 260
NotBoundException, 246
notifyActive method, 227
notifyDestroyed method, 87, 227
notifyIncomingMessage method, 385, 403
notifyPaused method, 87, 227
notifyStateChanged method, 127
NullPointerExecption, 169
numberOfSegments method, 380

O

obfuscation, 52
Object class, 26
object-oriented applications, 273
ODMs (original design manufacturers), 3
OEMs (original equipment manufacturers), 3
OMA (Open Mobile Alliance), 386
one-shot positioning requests, location-based services and, 501, 518
one-to-many protocols, 375
open method, 164, 173
  Connector class and, 295, 300, 378, 385
  HttpConnection class and, 311
Open Mobile Alliance (OMA), 386
openDataInputStream method, 166, 301
openDataOutputStream method, 166, 301
opening files, 166
openInputStream method, 166
openOutputStream method, 166, 314
openPIMList method, 176
openRecordStore method, 136
openStore method, 153

optimization, 52
option identifiers, 302
OPTIONS method, 307
Orientation class, 502
original design manufacturers (ODMs), 3
original equipment manufacturers (OEMs), 3
OutputConnection interface, 295
OutputStream class, 301, 313
OutputStreamWriter class, 30

P

packages, 8, 15
packaging
  MIDlets, 53–56
  Xlets, 75
paint method, 123–125, 239
  Container class and, 235, 256
  GameCanvas class and, 195
  Graphics class and, 235
  LayerManager class and, 202
  WeatherItem sample class and, 131
Panel class, 263
Panel container, 262
<PARAM>, 258
parse method, 362, 371
parts, MMS messages and, 376, 380, 382, 383
pauseApp method, 85, 86, 491
paused state, 479
  of MIDlets, 85
  of Xlets, 224
pauseXlet method, 224, 225, 230
payments, JSRs and, 541
PBP. *See* Personal Basis Profile
PDAs (personal digital assistants), 6
  CDC and, 12
  JSRs and, 541
permissions, 6, 319
  APDU and, 420
  granting for network connections, 327
  JCRMI and, 421
  location-based services and, 508
  MIDP and, 13
Personal Basis Profile (PBP), 12, 14, 223–252
  AGUI Optional Package and, 266
  developing lightweight user interfaces via, 233–241
  JSRs and, 541
  system properties of, 241
personal digital assistants. *See* PDAs
personal identification number (PIN), smart cards and, 420
personal information management. *See entries at* PIM
Personal Profile (PP), 12, 15
  applets, support for, 253
  JSRs and, 541
PersonalJava, 15, 23
PIM (personal information management), 161, 174–190, 438
  checking/ensuring availability of the PIM package, 175, 176
  JSRs and, 541
  MSA requirements and, 533
  removing entries and, 184
  sample applications and, 185–190
PIM class, 174
PIM database, 176
PIM records, 177–184
  creating, 183
  modifying, 182
  reading from PIM database, 177
PIMException, 175
PIMItem class, 175, 177
  adder methods for, 182
  setter methods for, 183
PIMList class, 174, 184, 533
PIN (personal identification number), smart cards and, 420
ping-pong buffering, 124
PitchControl interface, 458
pixels, reference pixels and, 206
plain text XML elements, indicated by < >, 339
PlainTagConnection class, 436
plaintext, 415
play method, 496

Player interface, 450, 451
PlayerListener interface, 451, 460
players, multimedia content and, 449
playerUpdate method, 460
playFromResource method, 492, 493, 496
playing audio/video, for multimedia content, 493
playing state, 479
playTone method, 466–469
point-to-area protocols, 375
pointer events, 123
pointerDragged method
  Canvas class, 123
  CustomItem class, 126
pointerPressed method
  Canvas class, 123
  CustomItem class, 126
pointerReleased method
  Canvas class, 123
  CustomItem class, 126
polling
  events, 197
  key states, 195
  keystrokes, 196
POST method, 307, 313, 319
PP. *See* Personal Profile
preamble, to XML documents, 339
prefectch method, 453
prefetched state, 453
preverify tool, 11, 52
prevFrame method, 206
previousRecord method, 143
priorities, 101
private record stores, 133
privileges, 6
  third-party certification and, 17
  WMA and, 386
processBlock method, 430
processBytes method, 430
processMouseEvent method, 240
processRequest method, 259
producer-consumer model, 259
profiles, 8, 12–15
ProGuard obfuscator, 52, 426
project configurations, 52
properties, for Xlets, 242
protocols, multimedia content and, 454
proximity tags, 431
ProximityListener interface, 502, 507
public-key cryptography, 423
pull parsers, 342, 365
push parsers, 342, 355
push registry, 54, 303
  alarms and, 91
  contactless communication and, 432
  incoming message, 387
PushRegistry class, 303, 387
PUT method, 307
putClientProperty method, 269

Q

QR Code (Quick Response Code), 440
QualifiedCoordinates class, 502, 506, 508, 518
Quick Response Code (QR Code), 440
quotes, single (‘ ’), 339

R

raster images, 470
RateControl interface, 458
RC4 (Rivest Cipher 4), 423
RC4 encryption, using Bouncy Castle API for, 429
read method, 140, 301
readBoolean method, 140
readByte method, 140
readChar method, 140
readChars method, 140
readDouble method, 140
readFloat method, 140
readInt method, 141
readLong method, 141
readNDEF method, 436
readShort method, 141
readUTF method, 141
readVisualTag method, 444

read*X* method, 305
realize method, 453
realized state, 453
receive method
  Connector class, 385
  MessageConnection interface, 379
receiving messages, 385
  MMS messages and, 398–407
  SMS messages and, 391–398
record IDs, 135, 138
  removing records and, 144
  retrieving records and, 142
  updating records and, 144
record stores, 133–160
  accessing records in, 138–146
  listening for changes in, 145
  obtaining information about, 137
  opening/closing, 136
  platform limitations of, 145
  removing, 137
  WeatherWidget sample application and, 146–160
recordAdded method, 145
recordChanged method, 145
RecordComparator interface, 143
RecordControl interface, 458, 465
recordDeleted method, 145
RecordEnumeration interface, 143, 153
RecordFilter interface, 143
RecordListener interface, 145
records, 133
  accessing in record stores, 138–146
  size limits, record stores and, 146
RecordStore class, 135, 173
RecordStoreException, 136, 137
RecordStoreFullException, 136, 141
RecordStoreNotFoundException, 136, 137
RecordStoreNotOpenException, 138, 141, 144
reference implementations (RIs), 527
reference pixels, 206
registerAlarm method, 91
registerConnection method, 303, 390
Registry interface, 246
releaseTarget method, 477
remotable objects, making available to other Xlets, 246
Remote class, 245
remote content, reading, 309
Remote interface, 243, 277
Remote Method Invocation. *See entries at* RMI
remote objects, 243, 275
  accessing, 249
  invoking, 286
remote procedure call (RPC) services, 333
remote services, 280, 285
RemoteException, 246, 251, 277, 282
remove method, 184, 202
remove ToDo method, 184
removeAddresses method, 383
removeCommand method, 106, 195
removeContact method, 184
removeEvent method, 184
removeFileSystemListener method, 169
removeLandmarkFromCategory method, 508
removeMessagePart method, 383
removeMessagePartId method, 383
removeMessagePartLocation method, 383
removeRecordListener method, 145
renameCategory method, 184
rendering multimedia content, 452–469
  controlling the process, 458–461
  starting the process, 454–457
  SVG images and, 474–484
repaint method, 123, 125, 235
repetition, multimedia content and, 468
representational state transfer (REST), 333
reset method, 306
resolution, multimedia content and, 467, 468
resources, Xlets for, 242
resources directory, 36
resources for further reading
  countermeasures, 416
  cryptography, 415
  DataSources, writing custom, 456
  MMAPI, 448
  SVG/SVG Tiny, 472
  XML, 336

REST (representational state transfer), 333
RESTful web services, 333, 356, 362
resume method (deprecated), 27
resumeRequest method, 87
revision-control software, 36
RFID devices, 436, 541
RFID tags, 431
RIs (reference implementations), 527
risks, security and, 414
Rivest Cipher 4 (RC4), 423
RMI (Remote Method Invocation), 273–287, 333
  advantages/disadvantages of, 276
  architecture of, 274
  JSRs and, 541
  Personal Basis Profile and, 14
  using, 280–286
RMI interfaces, 277, 282
RMI OP (RMI Optional Package), 278
RMI over Internet Inter-Orb Protocol (RMI-IIOP), 278
RMI server application (sample), 285
RMI stack, 243
RMI-IIOP (RMI over Internet Inter-Orb Protocol), 278
rmic compiler, 276, 281, 284
RMISecurityManager class, 277
rms interface, 133
root element, XML and, 336
root file systems, 169
rootChanged method, 168
round buttons, 236–240
RPC (remote procedure call) services, 333
rtp scheme, 455
rtsp scheme, 455
run method
  WeatherWidget sample application and, 318
  receiving messages and, 403
  SVG images and, 483
  media playback and, 491, 493
  location and, 518
run. sendMsg method, 397
Runnable interface, 318, 395
Runtime class, 27

S

SA (Selective Availability), 500
sample applications
  cat/butterfly game, 207–217
  ContactLoaderThread class and, 189
  Location class and, 146–150, 154
  LocationStore class and, 146, 150–155, 169–173
  media player, 484–496
  PIM package and, 185–190
  RMI server, 285
  WeatherApplet, 57–76, 281
  WeatherItem sample class and, 127–131
  WeatherWidget. *See* WeatherWidget
sample code. *See* source code
SATSA (Security and Trust Services API for J2ME), 416–425, 533
SATSA-CRYPTO API, 416, 422–425
  vs. Bouncy Castle API, 430
  encrypting/decrypting messages and, 423
  message digest creation and, 422
  MSA and, 533
SATSA-Java Card RMI (SATSA-JCRMI), 416
SATSA-public key infrastructure (SATSA-PKI), 416
SAX parser, 357, 360
Scalable 2D Vector Graphics. *See* SVGAPI
ScalableGraphics class, 474, 477
ScalableImage class, 474
scheduleMIDlet method, 95
schemes, MMAPI and, 454, 463
SCP (Secure Copy Protocol), 56
Screen class, 99, 114, 122
Screen Designer, 39–43
screen transitions, for WeatherWidget, 39
screens, 114–122
  JTWI recommendations for, 528
  MSA requirements for, 532
  MSA2 requirements for, 534
  for WeatherWidget, 41
SD (Secure Digital) card, 77
Secure Copy Protocol (SCP), 56
Secure Digital (SD) card, 77

security, 13, 413–445
  Java ME security model and, 6
  applets and, 253
  understanding the need for, 413–416
Security and Trust Services API for J2ME (SATSA), 416–425, 533
SecurityException, 141, 165
  LocationManager methods and, 505
  open method and, 296
  Player instances and, 463
  SMS messages and, 378
SecurityInfo interface, 326
seed method, 189
segments, SMS messages and, 380
Selective Availability (SA), 500
sending messages, 380–385
  MMS messages and, 398–407
  SMS messages and, 391–398
sendMMS method, 406, 495
sendMsg method, 396, 397, 406
serialization, 139, 150
ServerSocketConnection class, 302
service brokers, 332
service providers, 332
service requestors, 332
service-oriented architecture (SOA) services, 333
set-top boxes, 5
  CDC and, 12
  JSRs and, 540
setAddress method, 380
setAddressInfo method, 508
setAnimatedTile method, 205
setBinary method, 183
setBoolean method, 183
setCell method, 204
setChars method, 110
setCommandListener method, 103, 116, 195
setConditions method, 131
setConstraints method, 109
setCurrent method, 118, 199, 478, 483
setDate method, 110, 183
setDescription method, 508
setDoubleBuffered method, 270
setFont method, 108
setForecast method, 281
setFrame method, 206
setHeader method, 383, 385
setHidden method, 167
SetImage class, 404
setImage method, 118
setIndicator method, 118
setInitialInputMode method, 109
setInputMode method, 110
setInt method, 183
setItemCommandListener method, 106, 111
setItemSateListener method, 116
setLabel method, 107, 108
setLayout method, 107
setLocation method, 281
setLocationListener method, 506
setMediaTime method, 453
SetMessage class, 392, 396
setName method, 508
setPaused method, 216
setPayloadData method, 381
setPayloadText method, 380
setPreferredSize method, 107
setQualifiedCoordinates method, 508
setReadable method, 167
setRecord method, 138, 144
setRequestMethod method, 312, 314, 334
setRequestProperty method, 312, 336
setSelectedFlags method, 114
setSelectedIndex method, 114
setSocketOption method, 301
setStartContentId method, 382
setString method, 110, 118, 183
setStringArray method, 183
setSubject method, 383, 384
setText method, 108
setTimeBase method, 453
setTimeout method, 116, 118
setTransform method, 206
setup state, HTTP communication and, 312

setViewWindow method, 201
setWritable method, 167
set_location method, 517
shared objects, 243–251
  implementing, 244
  using, 249
shared record stores, 133
Shockwave Flash (SWF) format, 470
short codes, 374
Short Message Service - Cell Broadcast (SMS-CB), 375
Short Message Service. *See* SMS messages
showDocument method, 257
showException method, 492
showStatus method, 257
showSupportedMedia method, 491, 492
signing applications, 55
silence, multimedia content and, 468
Simple Object Access Protocol (SOAP), 332, 355
single quotes (' '), 339
SIP API for J2ME, 541
SIP protocol, MSA and, 534
size method, 114
skeletons, 276
smart cards, 416–421
smartcardslots property, 419
SMS aggregators, 374
SMS Center (SMSC), 374
SMS messages, 374
  encoding/decoding and, 381
  JSRs and, 541
  message size and, 374, 380
  segments and, 380
  sending/receiving, 391–398
SMS-CB (Short Message Service - Cell Broadcast), 375
SMSC (SMS Center), 374
SMSSender class, 392, 396
SOA (service-oriented architecture) services, 333
SOAP (Simple Object Access Protocol), 332, 355
SOAP-based web services, 355
socket communication, 300–303
socket options, 302
SocketConnection class, 300
soft keys, 270
solidus (/) character, 163, 165
source code
  applets, 255, 258, 265
  actionPerformed method, 61, 74
  alarms, 91
  alerts, configuring, 117
  ByteArrayInputStream, creating, 140
  ByteArrayOutputStream, creating, 139
  command objects/displayable objects, relationship between demonstrated, 103
  contacts, 184, 189
  content, downloading via ContentConnection, 309
  control loop, implementing, 198
  cookies, 335
  data, sending to server, 313
  DataSource, 455
  DataInputStream, creating, 140
  DataOutputStream, creating, 139
  device location, determining, 503
  encrypting messages, 424
  FCOP, interrogating the system for version of, 164
  forms, adding items to, 205
  frame sequences, 206
  GameCanvas class, 199
  "Hello World!", 304, 307
  Hello World applet, 256
  Hello World MIDlet, 83
  "Hello Xlet", 232
  HTTP connections, 295
  HTTP methods, specifying, 334
  HTTPS connections, 326
  images, sending in messages, 384
  imaging sensors, capturing snapshots from, 461
  indexes, array of, 203
  IXC mechanism, 247
  JAD files, 54

Java AWT components/containers, 256
Java SE stub classes, 284
lightweight components, 236–240
kXML parser, 366–372
key states, polling, 189
listConnections method (PushRegistry class), using, 389
Location class, 146, 343–355
Location interface, 282
location simulation, 519
LocationImpl class, 283
LocationStore class, implementing, 150, 170
LocationParser class, 371
LocationParserHandler class, 357–361
locators, for multimedia content, 455
media playback, stopping/reclaiming resources, 456
message digests, 422, 428
MIDlets, 43–50, 208
MIDlet-Push entries, hypothetical, 388
MMS message, 384
MMSMIDlet class, 399–404
MMSSender class, 404
MP3 player, 454
MultimediaMIDlet class, 486–492
NDEF-enabled targets, registering listeners for, 433
NDEF records, 437
paint method, 235
PIM package, 185
QR Codes, encoding/decoding, 441
random key generation, 430
RC4 encryption, using Bouncy Castle API for, 429
record fields, enumerating over, 180
record store, filtering, 143
RecordControl interface, 465
remote objects, invoking, 286
RESTful web service, 362
resources, accessing, 242
RMI server application, 285
root file systems, enumerating, 169
rootChanged method, overriding, 168
SAX parser, 361
sendMsg method, 495
ServerSocketConnection, obtaining, 302
setCell method, 204
SettingPanel class, 68
shared objects, 244, 249
smart cards, 419, 421
SMSMidlet class, 392
SMSSender class, 396
Sprite class, 206
SpriteCanvas class. 210
SpriteSampleMIDlet class, 209
StreamConnection interface, 300
supported media/protocol types, 457, 492
SVG images, 474, 475, 479, 480
SVG Tiny document, 470
TextMessage interface, setting/getting payload and, 380
to-do items, returning summary of, 180
toByteArray method, invoking, 140
ToneControl instances, 467, 468
TiledLayer class, 204
unparsed character directive, 340
video clips, playing, 459
WeatherApplet, 76
WeatherController class, 72
WeatherFetcher sample class, 315, 319–325, 362
WeatherWidget sample application, 155–160, 315, 319–325, 509–518
WeatherXlet, 62, 73
Web pages, obtaining, 306, 307
wireless messages, 438
Xlets, 225, 227
XML document, 336
XML namespaces, 340
Spacer class, 107
special characters, 339
Sprite class, 194, 201, 205–207
collision detection and, 207
transformations and, 206
SpriteCanvas class, 210–216

SpriteSampleMIDlet Class, 209
src directory, 36
start method, 254, 453
startApp method, 85, 86
  init method and, 491
  initialize method and, 95
  startReceive method and, 395
startCamera method, 462
startDocument event, 357, 360
started state, 453
startElement event, 357, 360
starting content ID, MMS messages and, 382
starting XML tag, indicated by </ >, 339
startReceive method, 395
startup events, 90–96
startXlet method, 224, 225, 229
states, Player instances and, 452, 460
static SVG images, 474
status codes for HTTP communication, 308
stop method, 254, 453, 456
stop method (deprecated in Java SE), 27
stopCamera method, 462
stopped state, 479
StopTimeControl interface, 459
StreamCipher interface, 429
StreamConnection interface, 295, 300
StreamConnectionNotifier class, 302
StreamConnectionNotifier interface, 295
String class, 28
StringBuffer class, 28, 342, 355, 360
StringItem class, 107, 108
stubs, 275, 284
subjects, MMS messages and, 383
Subversion (SVN), 36
suites (JAD/JAR pairs), 53
Sun Java Wireless Toolkit, 43, 53, 518
suspend method (deprecated), 27
SVG images, MSA requirements and, 533
SVG Tiny, 470
SVGAnimator class, 474, 477–480
SVGAnimatorWrapper class, 484
SVGAPI (Scalable 2D Vector Graphics), 470–496
  class/interface hierarchy of, 474
  JSRs and, 541
  modifying images and, 480
  organization of, 472
  rendering images and, 474–480
  sample media player and, 484–496
  using NetBeans and, 483
SVGCanvas, 477
SVGDocument class, 474, 483
SVGImage class, 474, 482
SVGImageItem class, 475
SVGMenu class, 484
SVGPlayer class, 484
SVGSplashScreen class, 484
SVGWaitScreen class, 484
SVN (Subversion), 36
SWF format (Adobe), 470
Swing, 267
  AGUI and, 270, 271
  user-interface model and, 99
SwingUtilities class, 269
symbologies, 440
SymbologyManager, 440, 443
System class, 27

T

tags, in XML, 336
targetDetected method, 433, 435
TargetListener interface, 433
TargetProperties, 433
TCKs (Technology Compatibility Kits), 527
TCP connections, 295, 300, 302
Technology Compatibility Kits (TCKs), 527
telematics, JSRs and, 541
telephony, JSRs and, 541
television, JSRs and, 541
tempo, playback and, 468
TempoControl interface, 459
test directory, 36
text messages. *See* wireless messaging
TextBox class, 99, 114, 119

TextField class, 109, 119
TextListener, 265
TextMessage interface, 376, 379, 380
theming, JSRs and, 541
third-party certification, 17
thread groups, 27
threats, security and, 414
TiledLayer class, 194, 201
  indexes and, 203–205
  sample game application and, 217
tiles, 194, 201
time APIs, 28
TimeBase interface, 451
Timer class, 29, 91
TimerTask class, 29, 91
timestamps, MMS messages and, 382
TimeZone class, 28
to-do items, 161, 174
  creating, 183
  PIMItem class and, 177
  removing, 184
  returning summary of, 180
toByteArray method, 140
toBytes method, 150, 154, 355
ToDoList class, 533
token smart cards, 417
ToneControl interface, 459, 466–469
tones, playing, 466–469
top-level windows, restrictions on, 269
totalMemory method, 27
toXml method, 343, 355
TRACE method, 307
transformations, Sprite class and, 206
traversal operations, 126
traverse method, 126, 131
truncate method, 167
trusted applications, 13
types, 101

U

UDDI (Universal Description, Discovery, and Integration), 332
UDPDatagramConnection class, 304
UI. *See* user interfaces
UnexpectedException, 246
unexportObject class, 284
UnicastRemoteObject class, 278, 279, 283
Uniform Resource Identifiers (URIs)
  discovering contactless targets and, 433
  smart cards and, 418
Universal Description, Discovery, and Integration (UDDI), 332
Universal Product Code (UPC), 440
Universal Subscriber Identity Modules (USIMs), 416
unmarshalling data, 342, 355
unparsed character directive, 339
unrealized state, 453
unregisterConnection method, 390
UnsupportedOperationException, 269
UPC (Universal Product Code), 440
update method, 235, 239, 318, 325, 425
updateLandmark method, 508
updateLocation method, 154
URFL device description repository, 300
URIs (Uniform Resource Identifiers)
  discovering contactless targets and, 433
  smart cards and, 418
urlEncode method, 319
URLs
  creating, 295
  syntax for, 297, 378
user interfaces (UIs), 97–131
  application screens for, 114–131
  items for, 104–112
  JSRs and, 541
  lightweight, developing via PBP, 233–241
USIMs (Universal Subscriber Identity Modules), 416

V

valid documents, XML and, 341
vector image format, 470
verifyPIMSupport method, 189
VeriSign, 55
versioning, record stores and, 138
video clips, playing, 459

VideoControl interface, 459, 461, 494
viewports, 477
visible items, 97–116
visual tag recognition, 432
visual tags (visual targets), 431–444
  encoding/decoding, 436
  JSRs and, 541
  recognizing/generating, 440–444
VisualTagCodingException, 444
VisualTagConnection interface, 432, 436, 440, 444
volume, playback and, 468
VolumeControl interface, 494

W

W3C, 470
WANs (Wide Area Networks), location-based services and, 500
WAP (Wireless Application Protocol), 341
WAP-209-MMSEncapsulation standard, 386
WAP Binary XML (WBXML), 341
waypoint/waypoints elements, 519
WBXML (WAP Binary XML), 341
WeatherApplet (sample application), 57–76
  packaging/executing, 75
  RMI OP and, 281
WeatherFetcher (sample) class, 315–325, 362
WeatherItem (sample) class, 127–131
WeatherWidget (sample application), 37–56, 315–325
  creating, 38–51
  FileConnection class, integrating into, 173
  integrating Location and LocationStore classes into, 155–160
  kXML parser and, 371
  Location API and, 509–518
  Location class and, 146–150, 155–160, 343–355
  LocationParserHandler class and, 357–361
  MMAPI/SVGAPI, 484
  records stores and, 146–160
WeatherWidget class, 518
Web pages, obtaining, 306
web services
  accessing, 331–372
  architecture of from client perspective, 331–341
  common Java ME programming tasks for, 342
  HTTP and, 306
  JSRs and, 541
  XML support for, 341–372
Web Services Description Language (WSDL), 332
webRequest, 423
well-formed documents, XML and, 341
white space, XML and, 339
Wide Area Networks (WANs), location-based services and, 500
Window class, 234, 263
Window container, 262
window toolkits, 230
  implementing your own components for, 234
  PBP and, limitations of, 240
Wireless Application Protocol (WAP), 341
wireless messaging, 373–407
  creating messages and, 379
  encrypting/decrypting messages and, 423, 429
  exchanging messages directly, 438
  JSRs and, 540, 541
  receiving messages and, 385, 391–398, 398–407
  sending messages and, 380–385, 391–398, 398–407
  WMA for, 375–387, 391–407
Wireless Messaging API. *See* WMA
wireless operators, Java ME market perspective and, 3, 4
Wireless Universal Resource File (WURFL), 17
WMA (Wireless Messaging API), 375–387
  JTWI requirements and, 529
  message encoding/decoding and, 381
  MSA requirements and, 533
  privileges and, 386

push registry and, 387
using, 391–407
versions of, 375
write method, 139, 301
writeBoolean method, 139
writeByte method, 139
writeChar method, 139
writeChars method, 139
writeDouble method, 139
writeFloat method, 139
writeInt method, 139
writeLong method, 139
writeNDEF method, 436, 437
writeShort method, 139
writeUTF method, 139
write*X* method, 305
WSDL (Web Services Description Language), 332
WURFL (Wireless Universal Resource File), 17

X

Xlet application model, backward compatibility for, 253
Xlet programming model, 14
XletContext interface, 225, 226, 242, 246
Xlets, 223–252
vs. applets, 254, 258
considerations for developing, 233
creating, 57–77, 225, 227–233
data sharing and, 243–251
dependencies and, 230–233
JAR files and, 242
life cycle of, 224
Mobility Pack for, 34, 57
properties/resources for, 242
states of, 224
XletStateChangeException, 225, 226
XML
for data representation, 336–341
generating in applications, 343–355
JSRs and, 541
MSA2 and, 536
sample document and, 336
web services and, 331, 334, 341–372
XML character entities, 339
XML files, Xlets and, 76
XML parsers, 342, 355, 356
XML schemas, 341
Xparse-J parser, 372

Y

YAML (YAML Ain't Markup Language), 331